PROGRAMMING AND UTILIZING DIGITAL COMPUTERS / Ledley / McGraw-Hill

171
THRIFTBOOKS
10 24346 171

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS

Robert S. Ledley
National Biomedical Research Foundation
and
The Johns Hopkins University



McGraw-Hill Series in Information Processing and Computers

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS

This is the first programming book to discuss thoroughly ALGOL, COBOL, programming for special purpose computers, and programming to achieve intelligence.

The author has divided his material, on both scientific and business levels, into three sections:

Part I discusses machine language programming, covering the three-, two-, and one-address instruction systems; indirect addressing and use of index registers; use of associative memories; variable word length addressing; concepts of threaded lists and push-down lists; programming special purpose computers, such as digital differential analyzer, logical control computer, array manipulation computer; and large-scale combined computing units.

Part II covers automatic programming languages, including complete chapters on ALGOL, COBOL, and modern ideas in programming to achieve intelligence including programming language translation, list processing, and heuristic programming.

Part III deals with methods for utilizing computers. Serving also as a source for programming problems, it is designed to orient the student in the techniques he will eventually be using in actual situations. A particularly unique aspect of the book, this part introduces computer-oriented numerical analysis; logic (propositional calculus); information processing and retrieval methods; self-correcting codes, including a new formula for calculating false drops.

(continued on back flap)

(continued from front flap)

Both ALGOL and COBOL automatic programming languages are treated thoroughly and lucidly. ALGOL is typical of other less general algebraic languages such as FORTRAN developed by various computer manufacturers. COBOL exemplifies business-oriented, data-processing, automatic-programming languages. These chapters are noteworthy in that they present for the first time in book form these internationally accepted automatic programming systems.

Also worthy of special attention are the chapters on programming for special purpose computers and programming to achieve intelligence. The latter chapter contains a description of automatic programming language translation; heuristic programming; programming to create music compositions; and programming to aid medical diagnosis.

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS will help fill the need for a modern, up-to-date college-level programming text. It contains extensive automatic programming methods and other material of current interest in the programming field. Over one-third of the book is devoted to automatic programming alone.

Note: *Electronic News* in its August 24, 1961, edition reported that all computers purchased by the Department of Defense after January, 1962, must have a working COBOL and ALGOL system. Since programmers will want to be up to date on these required automatic programs, the new Ledley book will furnish them with complete coverage of the systems.

Other McGRAW-HILL Books

DIGITAL COMPUTER AND CONTROL ENGINEERING

By ROBERT S. LEDLEY. *McGraw-Hill Series in Electrical and Electronic Engineering.* 835 pages, $14.50

A pioneer in the field of digital computers and digital controls. This book is a comprehensive elementary engineering textbook for advanced courses. Its unified treatment covers almost all aspects of the field from a detailed engineering point of view. The subjects covered were chosen to present a continuous, natural development of the major areas of digital computers and their controls.

PROGRAMMING AND CODING FOR AUTOMATIC DIGITAL COMPUTERS

By G. W. EVANS II, Stanford Research Institute; and C. L. PERRY, University of California, La Jolla. *McGraw-Hill Series in Information Processing and Computers.* 264 pages, $9.50

Techniques, methods, and facts essential to programming digital computers efficiently are brought together in this practical book. It outlines systematic procedures that enhance programming and coding skills and explains the computer's basic design so you can use it effectively. The book describes how the computer installation is established, organized, and operated for maximum performance.

COMPUTER HANDBOOK

By HARRY D. HUSKY and GRANINO A. KORN, University of Arizona. 1288 pages, $25.00

Gives detailed technical information and a wealth of industrial know-how on modern design practices in the field of analog-computer components and systems. Included are general-purpose analog computers and representative special-purpose machines. Many actual circuit diagrams are included as concrete examples of design principles or for direct adaptation to the designer's problems.

INTRODUCTION TO ELECTRONIC DATA PROCESSING EQUIPMENT—Its Operation and Control

By ROBERT V. OAKFORD, Stanford University. *McGraw-Hill Series in Information Processing and Computers.* 352 pages, $10.00

Emphasizing man-machine communications, this book explains how to communicate with electronic computers and auxiliary punched card equipment and discusses the principles of the equipment's operation. The book discusses existing rather than hypothetical equipment and languages. The IBM 650 is used as an example in the discussion of computer organization and machine language programming, since it is representative, widely used, and yet less complex than are larger computers.

McGRAW-HILL BOOK COMPANY, INC.

330 West 42nd Street — New York 36, N. Y.

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS / Ledley / McGraw-Hill

171
THRIFTBOOKS
10 24346 171

LEDLEY

Programming & Using Digital Computers

McGraw-Hill

McGraw-Hill Series in Information Processing and Computers

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS

McGraw-Hill Series in Information Processing and Computers

J. P. NASH, CONSULTING EDITOR

RICHARD W. HAMMING, ASSOCIATE CONSULTING EDITOR

DAVIS · Computability and Unsolvability

EVANS AND PERRY · Programming and Coding for Automatic Digital Computers

GOTLIEB AND HUME · High-speed Data Processing

KARPLUS · Analog Simulation

LEDLEY · Programming and Utilizing Digital Computers

MCCORMICK · Digital Computer Primer

OAKFORD · Introduction to Electronic Data Processing Equipment

ROGERS AND CONNOLLY · Analog Computation in Engineering Design

SCOTT · Analog and Digital Computer Technology

WILLIAMS · Digital Computing Systems

WRUBEL · A Primer of Programming for Digital Computers

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS

ROBERT STEVEN LEDLEY

NATIONAL BIOMEDICAL RESEARCH FOUNDATION
and THE JOHNS HOPKINS UNIVERSITY

WRITTEN WITH THE ASSISTANCE OF

JAMES BRUCE WILSON

National Biomedical Research Foundation

McGRAW-HILL BOOK COMPANY, INC.

New York San Francisco Toronto London

PROGRAMMING AND UTILIZING DIGITAL COMPUTERS

Copyright © 1962 by the McGraw-Hill Book Company, Inc. All Rights Reserved. Printed in the United States of America. This book, or parts thereof, may not be reproduced in any form without permission of the publishers. *Library of Congress Catalog Card Number* 62-10208

II

36982

TO MY MOTHER AND FATHER
KATE AND *JOSEPH*

PREFACE

Point of View. This book was written as a college text in programming digital computers; it can be used at various levels, ranging from sophomore to first-year graduate. The book is intended to fill the great need for an up-to-date, comprehensive text to provide an introduction to the many aspects of the digital-computer programming field, including automatic-programming languages as well as machine languages, the programming of special-purpose computers as well as general-purpose computers, mathematically oriented as well as business-oriented applications, and the modern areas of heuristic programming as well as mathematical decision programming. Of course, an introductory exposition of a field as large and rapidly advancing as this can never hope to treat all subjects exhaustively. Full treatment must be left to specialized source books and journal articles. Hence each chapter is designed primarily to introduce the student to certain fundamental concepts and techniques of development.

The book is divided into three parts, which consider respectively machine languages, automatic-programming languages, and data-processing techniques. I take the position that the use of machine languages encompasses the basic ingredients of programming, and hence Part 1 of the book is devoted to such languages. In addition to introducing the student to digital computers and some of their applications, Part 1 discusses a variety of instruction types and memory-addressing methods utilized in currently commercially available computers. However, many of the present-day practices and developments in the programming field are centered about the concepts of automatic programming. Thus I have devoted Part 2 of the book entirely to automatic-programming methods and applications, including an introduction to the algebraic language ALGOL (which is akin to FORTRAN in its basic aspects) and the business-oriented language COBOL, as well as a chapter intended to sketch some of the more modern ideas on programming to achieve intelligence.

Throughout the book the method of teaching is by example rather than by generalized exposition. I feel that the student can best grasp abstractions from specifics and develop skills through practice. Hence

Part 3 of the book is included primarily as a source of current methods and techniques in numerical analysis, logical analysis, and information handling, for use by the student to obtain more realistic material for problem analysis and programming practice. In addition, much attention has been paid to the exercises, of which more than 500 appear in the book. Almost every chapter ends with a bibliography, to stimulate the student in further reading in the field.

Outline of the Text. In Part 1 of the text, the first chapter is intended to *motivate* the student by delineating the wide range of applications of digital computers, and to *orient* the student with respect to the basic idea of a digital computer. The next two chapters are directly concerned with machine-language programming; their purpose is to expose the student to a large variety of instruction and operation formats, as well as the practice of coding. Sequences of instructions in three-, two-, and one-address-system instruction formats are considered, as well as the fundamental concepts of the loop, or iteration, and the subroutine. This is followed by a discussion of the various kinds of operations that instructions can involve, and by consideration of many types of indirect-addressing and relative-indexing memory-access methods. At the end of Part 1, in order to illustrate clearly the possibility of other than general-purpose computers, the digital differential analyzer is considered, along with real-time control and other special techniques. Also, a general discussion is included on the concepts underlying the "super" computers that have most recently been developed.

When first being introduced to automatic programming, the topic of Part 2, the student almost invariably becomes somewhat preoccupied with the question of how the generalized instructions can actually be realized in terms of a machine language. Such preoccupation interferes with further learning of more complicated automatic-programming methods. Hence Part 2 begins with a chapter (Chapter 5) designed to offset such problems by describing in a relatively concrete way how simulation routines, address-assigner routines, algebraic-compiling routines, and so forth, can be coded. In addition, the important concepts of threaded and push-down lists and their uses are introduced. Hence by the time the student reaches Chapter 6 (on ALGOL) he has gained at least some intuitive understanding of how automatic-programming languages can be coded, and he can then concentrate his attention more fully on the details of the languages themselves.

Chapter 6 is designed as an *introduction* to ALGOL, and should not be considered as a full exposition of that language. I do feel, however, that a knowledge of the contents of Chapter 6 will enable the student easily to assimilate the actual details of ALGOL, as well as FORTRAN and other algebraically oriented languages. For pedagogical reasons, I

have given a simplified version of ALGOL, stressing the more important concepts involved. To enhance the clarity of the exposition, I have taken the liberty of redefining some terms and changing some names (but I have tried to point out these changes and simplifications from ALGOL-60 in the footnotes). The symbolic metalanguage notation of the ALGOL-60 report† is introduced, for I feel that its use can enable a more precise, unambiguous definition of an automatic-programming language. Even if at first sight it may have a tendency to appear formidable to the student, I believe that with a little experience he will find it a simple and often invaluable notation. In Chapter 7, I consider the business-oriented language COBOL, again in a version simplified for pedagogical reasons. In order to avoid ambiguity, I here also utilize some of the metalanguage notation of the ALGOL-60 report. Although only the basic concepts of COBOL are considered, it is felt that with a knowledge of these, the student can easily digest the many detailed features of the actual COBOL language and other such data-processing languages. Since writing a program in the COBOL language really involves the use of an entire automatic-programming *system*, the chapter ends with the presentation of a complete example.

The title of Chapter 8, the last chapter of Part 2, is intended primarily to stimulate the student's imagination on some of the great potentialities that exist in the frontiers of programming research; however, I owe an apology to the programming teachers for this title, since quite obviously methods for achieving intelligence in computers are not described in the chapter. The theme of the chapter is the achievement on a computer of some realizations of the concepts of deductive and inductive inference. By *inductive inference* I mean *heuristic programming*, specific examples of which are presented. In our enthusiasm for the new and unusual, we must not, however, forget the more staid and relatively conventional methods of mathematical optimization, which I call *deductive inference*. Hence I also briefly sketch some of these methods and describe their application to computer-aided medical diagnosis. The chapter begins with a discussion of automatic-programming language translation, since the methods of syntactical analysis introduced there are used later on. The chapter ends with what is labeled as a consideration of programming to achieve abstraction and creativity, but which is, of course, simply a brief reporting of some ideas which have appeared in the literature and which profess to pertain to this topic. Because of the great wealth of material available in the literature, I have found it necessary to limit the discussion to only a few of the research approaches I considered "typical" or illustrative of some basic considerations—even so, because of the limited

† P. Naur et al., Report on the Algorithmic Language ALGOL 60, *Communs. ACM*, vol. 3, no. 5, pp. 299–314, May, 1960.

space that could properly be devoted to topics of this kind in an introductory text, it was necessary to be very sketchy. The entire chapter is calculated to raise more questions than it answers, but if it stimulates the student to think further about these advanced topics, it will have served its purpose. Certainly a text of this type would be incomplete without such a chapter!

Part 3 of the book is primarily included as a practice source in computer utilization techniques. The first chapter in this part, Chapter 9, considers elementary aspects of numerical analysis, stressing the concept of the polynomial-approximation approach for function evaluation and briefly touching upon ordinary and partial differential equations. The chapter concludes with a discussion of accuracy and error in numerical calculations. Chapter 10 considers computational methods in and applications of Boolean algebra. I believe that the role of Boolean algebra as an important tool and practical aid to the *computational analysis* of a large variety of problems (in fields other than switching-circuit theory and the logical design of digital computers) is not so well recognized as it might be. In Chapter 10, first Boolean algebra is introduced in terms of the propositional-calculus representations and the class, or set, representations. The method of using bases and associated designation numbers (which was first fully developed by the author, although it appeared implicitly in some earlier writings) has been found admirably suited to the teaching of the more complicated methods in Boolean algebra, such as the several modern procedures for the simplification of Boolean functions, the introduction of constraints into a problem, and the handling of logical dependence and independence. The final three sections of this chapter present methods which with a few exceptions are based on the author's original research. Here Boolean matrices are introduced and utilized for obtaining systematic computational methods for the solution to Boolean equations of a variety of types. Illustrations are then given of the application of these methods. The final chapter of Part 3, Chapter 11, is concerned with processing methods other than those used in numerical or logical analysis, which are, nevertheless, of fundamental importance in business and other activities requiring large-scale information handling and data reduction. In addition to techniques for searching and sorting, examples of methods for redundant (error-correcting) and irredundant (overlapping) coding are also considered.

Possible Course Structures. This text can be used in courses at the sophomore, junior, or senior college level or at the first-year graduate level, in several different ways—as a one-, two-, or three-semester course. Although undoubtedly each instructor will have his own personal preferences, Table 1 suggests the chapters and/or sections that could be covered

in such courses. I believe that even in an introductory (one-term) sophomore course, a good deal of automatic programming should be included, so that the student may view the concepts of the modern programming field with better perspective. It may be wise also to introduce the junior or senior to other than general-purpose programming, through a consideration of the digital differential analyzer. In the more leisurely

TABLE 1. SUGGESTED COURSE OUTLINE

Year	One-term course	Two-term course		Three-term course		
		Term 1	Term 2	Term 1	Term 2	Term 3
Sophomore	Chaps. 1, 2; Secs. 3-1 to 3-3, 3-7; Chap. 6; Secs. 8-1 to 8-3	Chaps. 1 to 3; Secs. 4-2, 4-3	Chaps. 5 to 7; Secs. 8-1 to 8-3	Chaps. 1 to 3; Secs. 4-2, 4-3	Chaps. 5 to 7; Secs. 8-1 to 8-3	Secs. 4-4 to 4-6; Secs. 8-4 to 8-8; Chaps. 9, 11
Junior	Chaps. 1, 2; Secs. 3-1 to 3-3, 3-7; Sec. 4-2; Chaps. 5, 6; Secs. 8-1 to 8-3	Part 1†	Part 2‡	Part 1†	Part 2‡	Part 3§
Senior	Chaps. 1 to 3; Sec. 4-2; Chaps. 5 to 7; Secs. 8-1 to 8-3	Part 1†	Part 2‡	Part 1†	Part 2‡	Part 3§
First year graduate	Chap. 1 (for home study); Chaps. 2 to 4; Chap. 5 (for home study); Chaps. 6 to 8	Part 1;† Chaps. 9, 11	Chap. 10; Part 2‡			

† Part 1 includes Chaps. 1 through 4.
‡ Part 2 includes Chaps. 5 through 8.
§ Part 3 includes Chaps. 9 through 11.

two- and three-term courses, further attention can be devoted to special-purpose computers. In the three-term courses, the third term might be given concurrently with the second term as a course entitled "Introduction to the Utilization of Digital Computers." *An alternative* to the suggestions given in Table 1 is a *one-term course* entitled "Introduction to Automatic Programming," where Chapter 1 could be covered (if the students have had no other association with digital computers) and then

immediately Chapters 6, 7, and 8 could be covered. The advantage of such a course is that in a single term the student would gain knowledge enabling him to grasp thoroughly automatic-programming languages used on commercially available computers, without having to wade through the details of machine-language programming; on the other hand, the disadvantage is obvious, that the student would be required to accept on faith alone the fact that such automatic-programming languages can be realized in terms of machine-language instructions.

Special Technical Features. Parts 1 and 3 of this book are basically excerpts of material in the author's engineering text.† Though this previous text included much material of value to programmers, the publisher felt that programmers would not be particularly interested in detailed methods for the logical and electronic-engineering design of a computer. These parts of the engineering text were eliminated to make room for a greatly expanded treatment of the subject of primary concern to the modern computer programmer—automatic programming. Chapters 1 to 4 of the present book correspond, with some rearrangements and deletions of material, to Chapters 1 to 4 and 8 of the engineering book but include a new discussion of modern memory-access methods. To Chapter 5 of the engineering book I have added a section on data-handling routines, including a discussion of threaded and push-down lists; also the discussion of algebraic compiling routines has been largely rewritten. The present chapter on numerical analysis includes a new discussion of "differences" and partial differential equations. Chapter 10 of this text is a greatly condensed version, for programmers, of the material contained in Chapters 10 through 14 of the engineering text, with the primarily engineering applications eliminated. Chapters 6 to 8 are entirely new, presenting material which to my knowledge appears in book form for the first time.

Special features of Part 1 include the use of a four-address system for introducing the basic ingredients of an instruction, and the discussion in Section 3-5 of the relative, indirect, augmented, and compound memory-addressing techniques. The specific method of Section 4-3 for coding the control computer was developed by the author, and the method for coding the logistics and business computer, appearing in Section 4-5, is based on the work of W. H. Marlow.

In Part 2, I present an introduction to the two automatic-programming languages that are becoming internationally accepted: ALGOL and COBOL. Sponsored by the Association for Computing Machinery and several foreign societies, the universal acceptance of ALGOL would undoubtedly have a great beneficial effect on the national and international exchange of ideas and methods of programming. The discussion of ALGOL also contains the primary features of other automatic programs, such as FORTRAN, GOTRAN, FOR TRANSIT, Honeywell Algebraic Compiler, Philco 2000 ALPAC, and so forth.

† R. S. Ledley, "Digital Computer and Control Engineering," McGraw-Hill Book Company, Inc., New York, 1960.

That the general acceptance of COBOL could produce extensive savings in the accounting and legal applications of computers has received wide recognition in business and industrial circles. In fact, some agencies of the United States government are requiring the preparation of a COBOL language by commercial computer manufacturers for computers to be used for such purposes.

Also included in Part 2 are several special features: the discussion of algebraic compilers (Section 5-8), based on the work of R. W. Floyd; the description of automatic-programming translation (Sections 8-2, 8-3), based on an idea of E. T. Irons and given in a formulation by the author and J. B. Wilson; and the treatment of computer aids to medical diagnosis (Section 8-5), based on the work of the author and L. B. Lusted. The discussion of heuristic programming and the general problem solver (Sections 8-6, 8-7) is based on articles by H. L. Gelernter and N. Rochester and by B. A. Newell, J. C. Shaw, and H. A. Simon. The utilization of the automatic-programming routine and the general problem-solver routine for concept recognition (Section 8-8) is the result of the author's original investigations, and the discussion of creativity is based on parts of a book by L. A. Hiller and L. M. Isaacson.

Special features of Part 3 include J. H. Wegstein's general method for accelerating the convergence of iterative solutions to equations (Section 9-2), the author's Tabledex method, and his techniques for searching with relaxed conditions (Sections 11-2, 11-4). A new, simpler version, due to W. R. Smith, of the exact formulas for "irredundant" (overlapping) coding has here (Section 11-7) replaced the more cumbersome formulas given in the engineering text. The computational methods for constraints in a Boolean algebra, logical dependence and independence, the solution to Boolean equations, the method of antecedence and consequence solutions, and the fundamental Boolean matrix formulas given in Chapter 10 are the author's original developments. W. R. Smith developed the general solution to the logical matrix equation (Section 10-14). J. B. Wilson contributed the decimal simplification method (Section 10-7), and R. D. Elbourn deserves credit for the direct method for finding prime implicants (Section 10-8).

Acknowledgments. It gives me the greatest pleasure to acknowledge the help and cooperation I have received from many persons during the preparation of this book. My sincerest thanks go to James Bruce Wilson, who carefully and tirelessly edited the entire manuscript, contributing many important and valuable improvements to the style and technical

presentation. Many colleagues helped by supplying advice and information, making constructive criticisms, and rendering assistance. Of these, the author particularly wishes to mention Joseph Blum, J. H. Wegstein, H. L. Hoffman, L. S. Rotolo, E. K. Pritchard, B. J. Dahl, V. Henriques, and D. Cohen. The figures in Chapter 1 are in large part due to the skill and talent of Joseph E. Ozefovich. Eva March Cuddy, Sheila Schlossenberg, and Doris I. Barker typed most of the manuscript and contributed secretarial assistance. And I wish to thank my wife, Terry Ledley, for assisting with the editing and also for her encouragement, patience, and lost weekends.

Robert Steven Ledley

CONTENTS

Preface vii

PART 1 MACHINE LANGUAGES

Chapter 1. Applications and Description of a Digital Computer ***1***

1-1. Introduction 1
Another Great Evolutionary Step in Mankind's Capabilities. Historical Notes. The Role of the Programmer. Chapter Outline.

1-2. Numerical Solution to Equations 8
Motion of Charged Particles in the Earth's Magnetic Field. Solutions to Equations in Biomedical Research.

1-3. Process Control 12
Machine-tool Control. Control of Chemical Plant.

1-4. Simulations 15
Industrial Dynamics. Man-Machine Simulation System.

1-5. Data Processing 20
Inventory Control. Airline Reservation System. Aids to Medical Diagnosis.

1-6. Block Diagram of a Computer 26
The Computer. The Memory and Computing Unit. Analogy between Hand Computation and the Functions Performed by the Parts of a Digital Computer. The Control. Input-Output. Summary.

1-7. Functional Description of a Computer 31
Words. Instructions and Addresses. The Accumulator. The Instruction Register and the Current-address Register. The Four Phases. Summary.

1-8. Words and Pulses 35
Binary Numbers and Bits. Words as Pulse Trains. Time Synchronization. Example of Adding. Instruction Decoding.

1-9. Input, Output, and Memory Systems 40
Input Information. Other Input-Output Methods. Memory Systems.

1-10. Exercises 48
References 49

Chapter 2. Principles of Programming ***53***

2-1. Introduction 53
Stages in Programming. Philosophy of Programming.

2-2. Number Systems: Conversion 54
Number Systems and the Radix. Conversion from One Number System to Another. Converting from Binary to Decimal.
2-3. Number Systems: Arithmetic 61
Addition and Multiplication Tables. Arithmetic Operations
2-4. Sequences of Instructions 62
Instruction Format. Octal Shorthand. The Computer Manual. Example. The Use of Temporaries.
2-5. Decisions 67
The Comparison Instruction. Example. The Sign Bit
2-6. Flow Charting and Symbolic Code Aids 69
Steps in Writing a Code.
2-7. Recursion Codes and Instruction Modification 72
Loops in the Flow Diagram. Instruction Modification. Instruction Modification in Loops. Loops within Loops.
2-8. The Subroutine 80
Use of a Subroutine. Example. Library of Subroutines.
2-9. Exercises 85

Chapter 3. Instruction Formats and Operation Types ***88***

3-1. Three- and Two-address Instruction Systems 88
Using Fewer than Four Addresses. The Three-address Instruction. Examples. The Two-address Instruction. Examples.
3-2. One-address Instruction Systems 93
The One-address Instruction. Examples. One-plus-one-address Systems.
3-3. Decimal Systems 96
Decimal-coded Binary. Example.
3-4. Arithmetic, Logic, Decision, and In-Out Instruction Types. . . 98
Floating Point. Arithmetic Instruction Types. Logical and Bit-handling Instruction Types. Decision Instruction Types. Read and Write Instruction Types.
3-5. Indirect Addressing and Relative Indexing 106
Variable Word Addressing. Indirect and Augmented Addressing. Relative and Compound Addressing. Recursion, or Loop-aiding, Instructions.
3-6. Special Coding Techniques 112
Multiway Branch. Switch Sensing. Input-Output. Table Look-up.
3-7. The Control Panel. 118
Purpose of the Control Panel. Lights and Signals. Buttons and Switches.
3-8. Exercises 122
References 124

Chapter 4. Programming for Special-purpose Digital Computers. ***126***

4-1. Introduction 126
Outline.
4-2. The Digital Differential Analyzer 127
Purpose. The Concept of the Digital Differential Analyzer. Computing Units. Programming.

4-3. Real-time Logical Systems Control: A Real-time Control Computer 132
An Example. Programming Logical Control. Description of a Special-purpose Logical-control Computer.
4-4. Special-purpose Information-retrieval Computer 137
An Example. A Small Retrieval Computer. Integration with a General-purpose Computer.
4-5. Manipulations with Rectangular Arrays: A Business and Logistics Computer 140
The Need. Basic Operations and Instruction Format. Illustrations of Coding. Description of a Special-purpose Business or Logistics Computer.
4-6. The Large-scale Data Processor. 150
Special Input-Output Problems. The Large-scale Data Processor.
4-7. Exercises 154
References 156

PART 2 AUTOMATIC-PROGRAMMING LANGUAGES

Chapter 5. Development of Automatic Programming ***159***

5-1. Introduction 159
5-2. Program Debugging Methods and Routines 160
The Problem of Program Debugging. Dynamic Program Debugging. Use of the Automonitor Program. The Automonitor Program Itself. Post-mortem Program Debugging.
5-3. Mathematical Interpretive Routines 164
Automatic Programming. Use of Mathematical Interpretive Routines. The Interpretive Routine Itself.
5-4. Simulational Interpretive Routines. 168
Use of Simulational Interpretive Routines. The Interpretive Routine Itself.
5-5. Data-handling Interpretive Routines 172
Use of Data-handling Routines. The Conventional Search. Close Up and Make Space. Threaded Lists. Push-down Lists.
5-6. Memory Space, Speed of Computation, and Automatic Programming 179
Sharing of the High-speed Memory. Complications. Advantages of the Compiler Routine.
5-7. Translator and Address-assigner Compiling Routines 182
Kinds of Compiler Routines. Translator. Writing the Translator Compiler Routine. Address Assigner.
5-8. Algebraic Compiling Routines 186
Decoder of Algebraic Symbols. Writing the Algebraic Compiler. Recursive Compiler.
5-9. Exercises 191
References 192

Chapter 6. Algebraically Oriented Languages: ALGOL ***193***

6-1. Languages 193
Introduction. Characters, Words, Expressions, and State-

ments. The Syntactical Definitions. The Semantic Discussion. The Grammatical Forms.
6-2. Example of an Algebraically Oriented Language: ALGOL . . . 202
Purpose. Example of the Language. Characters. Expressions.
6-3. Boolean, Arithmetic, and Designational Expressions. 208
Boolean Expressions. Arithmetic Expressions. Designational Expressions.
6-4. Statements. 215
Compound Statements and Blocks. Assignment Statements. ***go to*** Statements. Conditional Statements. ***for*** Statement.
6-5. Type, ***Array,*** and ***Switch*** Declarations 225
Need for Declarations. Type and ***Array*** Declarations. ***Switch*** Declarations. Grammar of Blocks: Local and Nonlocal Variables.
6-6. Procedure Statements and Declarations 228
Need. Procedure Statements. Procedure Declarations. Examples of Procedure Declarations. Functions and Procedure Statements.
6-7. Exercises 235
References 238

Chapter 7. Business-oriented Data-processing Languages: COBOL **240**

7-1. Common Business Oriented Language COBOL 240
Introduction. Characters and Words. Over-all Organization. Procedure and Data Divisions.
7-2. Operational Statements 247
Boolean and Arithmetic Expressions. Assignment Class of Statements. **GO TO** and **ALTER** Statements. Conditional Statements. ***for*** Statement. Procedure Statements and Declarations.
7-3. Over-all Data Organization 257
Orientation. Flow of Information during Processing. The Record Description.
7-4. Detailed Data Descriptions 263
Description of Physical and Operational Characteristics. Special-purpose Data Descriptions. The Working-storage Data Description.
7-5. File and Environment Descriptions. 271
The File Description. The File Data. Label Records. The Environment Description.
7-6. Data Handling by COBOL 277
Orientation. Data-handling Statements. Data-editing Clauses. File- and Record-handling Statements. A Complete Example.
7-7. Exercises 289
References 291

Chapter 8. Programming to Achieve Intelligence **295**

8-1. Advanced Concepts in Automatic Programming 295
Achieving Intelligence. List Processing.

8-2. Automatic-programming-language Translation 298
Syntax-directed Automatic-programming-language Translation. The Syntactical and Semantic Descriptions. The Translator.
8-3. Examples of Translating ALGOL Codes into a Symbolic Machine Language. 311
The Source and Target Languages for the Examples. Examples of Translation. Example of a More Efficient Syntax and Semantic Description.
8-4. Deductive Inference: Mathematical Optimization 324
Modern Mathematical Decision Theories. Linear Programming: Decision under Certainty. Game Theory: Decision under Uncertainty. Dynamic Programming.
8-5. Deductive Inference: Programming for Medical Diagnosis . . . 334
Reasoning in Medical Diagnosis. Listing the Diagnostic Possibilities. Determining Further Tests. Calculating the Alternative Diagnostic Probabilities. Bayes' Formula. The Dynamic Diagnostic-treatment Problem.
8-6. Inductive Inference: Heuristic Programming 348
Meaning of Heuristic Programming. Proving Geometric Theorems. The Heuristic-programming Technique. Programming Organization.
8-7. Inductive Inference: The General Problem-solving Program . . 355
Concept of a General Problem-solving Program. Example of Proving Trigonometric Identities. The Heuristic Method.
8-8. Abstraction and Creativity 362
Introduction. Deductive Methods in Abstraction. Inductive Methods in Abstraction. Formulation of Abstractions. Creativity.
8-9. Exercises 375
References 377

PART 3 DATA-PROCESSING TECHNIQUES

Chapter 9. Fundamentals of Numerical Analysis **383**

9-1. Simultaneous Linear Equations. 384
The Problem. The Method. Example.
9-2. Algebraic and Transcendental Equations 387
Successive Approximations. Accelerating the Convergence of Iterative Processes.
9-3. Function Evaluation: Series and Continued Fractions 393
Series Approximations. Continued-fraction Approximations. Differences of a Polynomial. Error Propagation of Differences.
9-4. Function Evaluation: Interpolation. 398
Polynomial Interpolation. Aitken's Method.
9-5. Function Evaluation: Best-fit Polynomial Approximation . . . 402
Polynomial Approximations. Best-fit Polynomials.
9-6. Integration. 404
By Definition. By the Trapezoid Rule. By Polynomial Approximations.
9-7. Differentiation. 407
By Definition. By Polynomial Approximation.

9-8. Undetermined Coefficients 409
The Method.
9-9. Differential Equations 410
Meaning of Solution. By Polynomial Approximation. Example. By Runge's Method.
9-10. Partial Differential Equations 414
Difference Quotients. By Iteration of Difference Equations. By Relaxation of Difference Equations.
9-11. Accuracy and Error 421
Sources of Error. Significant Figures and Round Numbers. Relative Accuracy and Absolute Accuracy. Significant Figures in Arithmetic Operations. Pitfalls.
9-12. Exercises 429
References 433

Chapter 10. Boolean Algebra ***435***

10-1. Introduction 435
Historical Background.
10-2. Definition of Logic and Propositions 436
Symbolic Logic. The Proposition. The Propositional Operations. Boolean Functions.
10-3. Boolean Expressions 439
Implication. Equivalence. Tautology. Equations of Boolean Algebra. Algebraic Proofs. Other Logical Operations.
10-4. Representations of Boolean Expressions 442
Sets. Intersection and Union. Logical Translation of English. Other Representations.
10-5. The Designation Numbers 447
The Basis. Determining the Designation Number of a Boolean Function. Important Results.
10-6. Forms of Boolean Functions 451
From Designation Number to Boolean Function. The First Canonical Form. The Second Canonical Form. Mongrel Forms.
10-7. Simplest Sum-of-products Representation 454
The Role of the Prime Implicants. The First Stage—Generating the Prime Implicants. The Second Stage—Using the Prime-implicant Chart. Summary. Decimal Method for Simplification.
10-8. Obtaining Essential Prime Implicants Directly 460
Preliminary Notational Review. Finding the Essential Prime Implicants Directly. Systematic Method of Procedure. Examples and Discussion.
10-9. Constraints 467
The Problem. Determining the Constrained Basis. Designation Numbers with Respect to a Constrained Basis. Constrained Boolean Algebra.
10-10. Logical Dependence and Independence 472
Logical Independence. Logical Dependence.
10-11. Boolean Matrices 474
Preliminaries: Boolean Matrix Operations. Solutions to

Boolean Matrix Equations. The Unitary Matrix Equation. Solutions to the Unitary Matrix Equation.
10-12. Boolean Transformations and the Fundamental Formulas . . . 479
Boolean Algebraic Transformations. The Fundamental Formulas. Boolean Algebraic Equations.
10-13. Application to Biochemical and Intelligence Problems 483
Logical Problems. Analyzing Experimental Results. Analyzing Intelligence Reports.
10-14. Exercises 487
References 493

Chapter 11. Searching, Sorting, Ordering, and Codifying ***495***

11-1. Introduction 495
Ingredients and Definitions. Generalized Concept of the Searching Problem in Terms of Boolean Algebra.
11-2. Methods of Searching 500
The First Two Methods. The Tabledex Method. Population vs. Substantive Ordering of the Lists.
11-3. Manual Searching Methods 504
Marginal-punch Cards. Peek-a-boo Cards. Programming a Computer to Generate a Manual Tabledex Collection.
11-4. Searching with Relaxed Conditions 506
The Problem of Relaxed Conditions. Searching for Items Associated with Any r Out of n Characteristics. Searching with Permanent Conditions. Searching with Preferential Conditions. Relaxing Sets of Conditions.
11-5. Sorting and Ordering 510
Minimum- or Maximum-in-pass. Transposition. Ordering by Radix Coefficients.
11-6. Ordering by Merging 515
Merging. Variations on Merging. Comparison of Ordering Methods.
11-7. Codifying: Error Correction and Superimposition 521
Definitions. Efficiency of Codifications. Self-correcting Redundant Codification. Superimposed Irredundant Codification.
11-8. Exercises 532
References 534

Appendix 537
Honeywell 800: Example of a Three-address Instruction System. Univac 1103A Scientific: Example of a Two-address Instruction System. IBM 7090: Example of a One-address Instruction System.

Name Index 545

Subject Index 549

PART 1

MACHINE LANGUAGES

CHAPTER 1

APPLICATIONS AND DESCRIPTION OF A DIGITAL COMPUTER

1-1. Introduction

Another Great Evolutionary Step in Mankind's Capabilities. The introduction of agriculture revolutionized ancient man's social, economic, and cultural potential. This was the first great step in the evolution of civilization. The more recent Industrial Revolution, vastly increasing man's productive capabilities, was the next great step and brought forth our present highly mechanized economic and interdependent social civilization. We are now on the threshold of still another new kind of revolution, based on machines that greatly increase man's "thinking" capabilities of planning, analyzing, computing, and controlling. Even though the digital-computer field has existed for scarcely a decade, many thousands of computers are already in daily use, penetrating almost all phases of our modern society, from nuclear-energy production and missile design to the processing of bank checks, business invoices, and medical diagnoses. This new and unique potential of mankind will have a marked influence not only on the physical sciences and technology but, perhaps even more significantly, on the biological sciences and the economic, political, and social aspects of our civilization. The changes that will evolve from this newest of mankind's potentialities are difficult to foresee, but they can be even greater than the spectacular changes occurring in our civilization from 1860 to 1960—from the beginnings of the Industrial Revolution to modern times.

The concept of an evolutionary development of civilization as a continuation of biological evolution has been emphasized by the noted biologist Julian Huxley and can be clearly seen in the great dependence and reliance of modern industrial and technological capabilities upon the heritage of past generations. The development of computer capabilities strongly displays such evolutionary characteristics. The use of computers to design more powerful computers has already begun. Hardly a

calculation is presently planned that does not utilize some previously written computer program. And so it is also with the evolution of concepts and ideas for utilizing computers in more and more complex ways, to handle such complicated control problems as factory, aircraft, or defense-systems control, to prove mathematical theorems, to simulate complicated many-variable situations, such as military tactics and strategy or economic planning and development, etc. To quote Julian Huxley:†

> If we hope to sum up the situation epigrammatically, we can say that, with the advent of man, conscious purpose was able for the first time to exert an operative effect on evolution; and that through the new knowledge acquired during the last hundred years, the evolutionary process was able for the first time to become conscious of itself. The next decisive step in evolution will accordingly be the fuller development of that self-consciousness. In particular it will be the conscious clarification of the future in the light of the long evolutionary past and of the study of human potentialities. Once this has been done, greater realization of possibilities rather than higher productivity, fulfillment rather than efficiency, will become the overriding aim.

Historical Notes. The first digital electronic computer with a high-speed electronic memory was successfully operated a little more than ten years ago. However, the concept of a computer dates back many centuries. John Napier's celebrated "bones," described in 1617, assisted in multiplication. In 1642 (the very year of Newton's birth), Pascal invented a mechanical computer that utilized a carry for adding; and in 1671 Leibnitz described a machine that could multiply by successive additions. The Jacquard loom (1801), which was evidently an outgrowth of several inventors' contributions, is memorable in that it utilized punched cards to control various weaving patterns. But Charles Babbage is probably the one pioneer most deserving of credit for developing many of the most important basic ideas utilized in modern computer concepts. Between 1834 and 1854 he worked on his "analytic engine," which utilized gears for the calculations; the machine was, however, never completed. These are some of the ancient historical roots of the computer field, which led eventually to the conventional desk calculating machine and the common punch-card machine (see Fig. 1-1).

It was not until the early 1940s that there arrived the true precursors of the modern computer. A series of relay computers were made at the Bell Telephone Laboratories and at Harvard University. The ENIAC, built at the University of Pennsylvania's Moore School of Electrical Engineering from 1942 to 1945, was the first computer to utilize elec-

† Julian Huxley, Evolution and Genetics, in James R. Newman (ed.), "What Is Science?" p. 289, Simon and Schuster, Inc., New York, 1955.

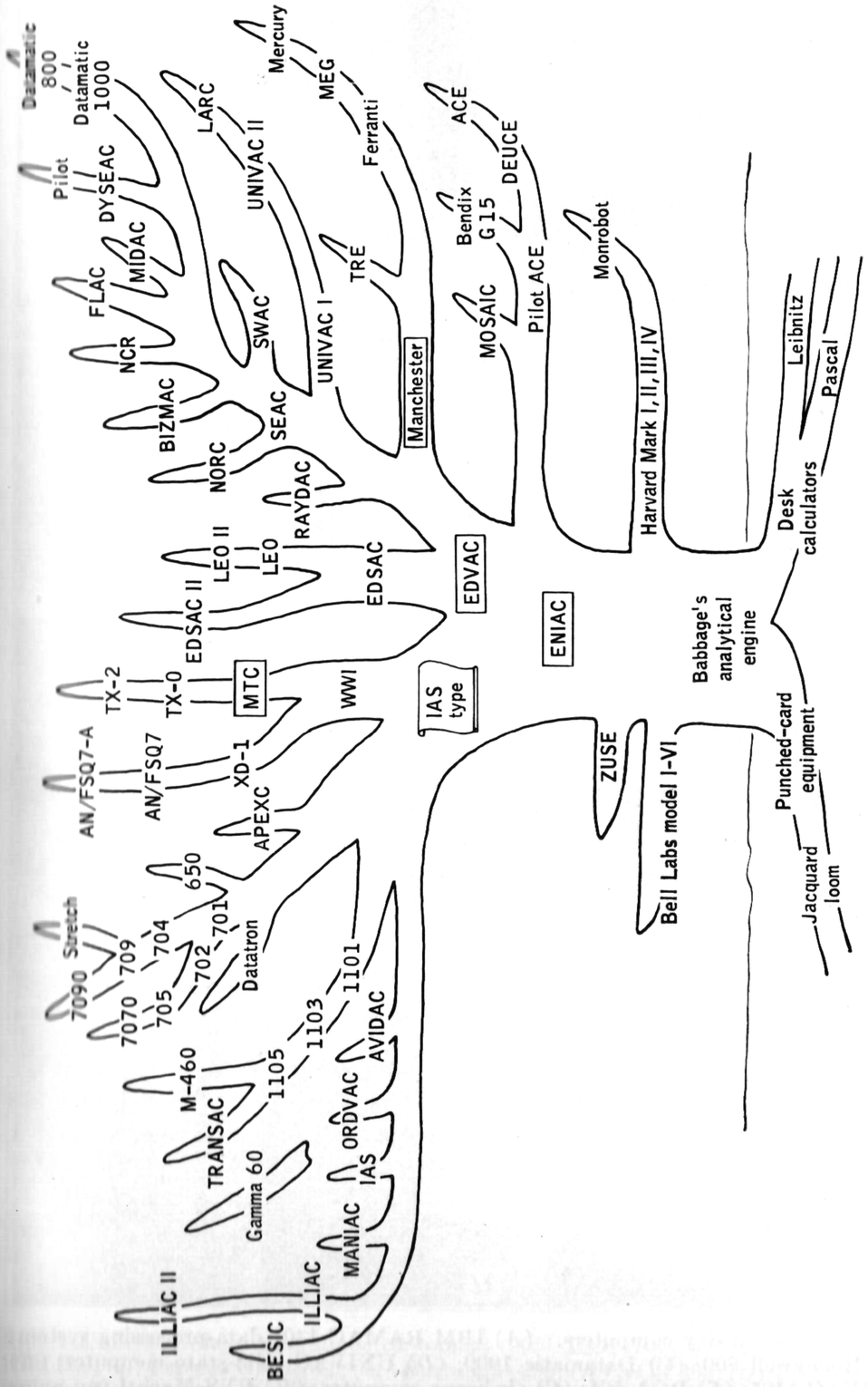


FIG. 1-1. The evolutionary tree of digital computers. We cannot claim that this tree is complete, but we hope that the major trends are reflected.

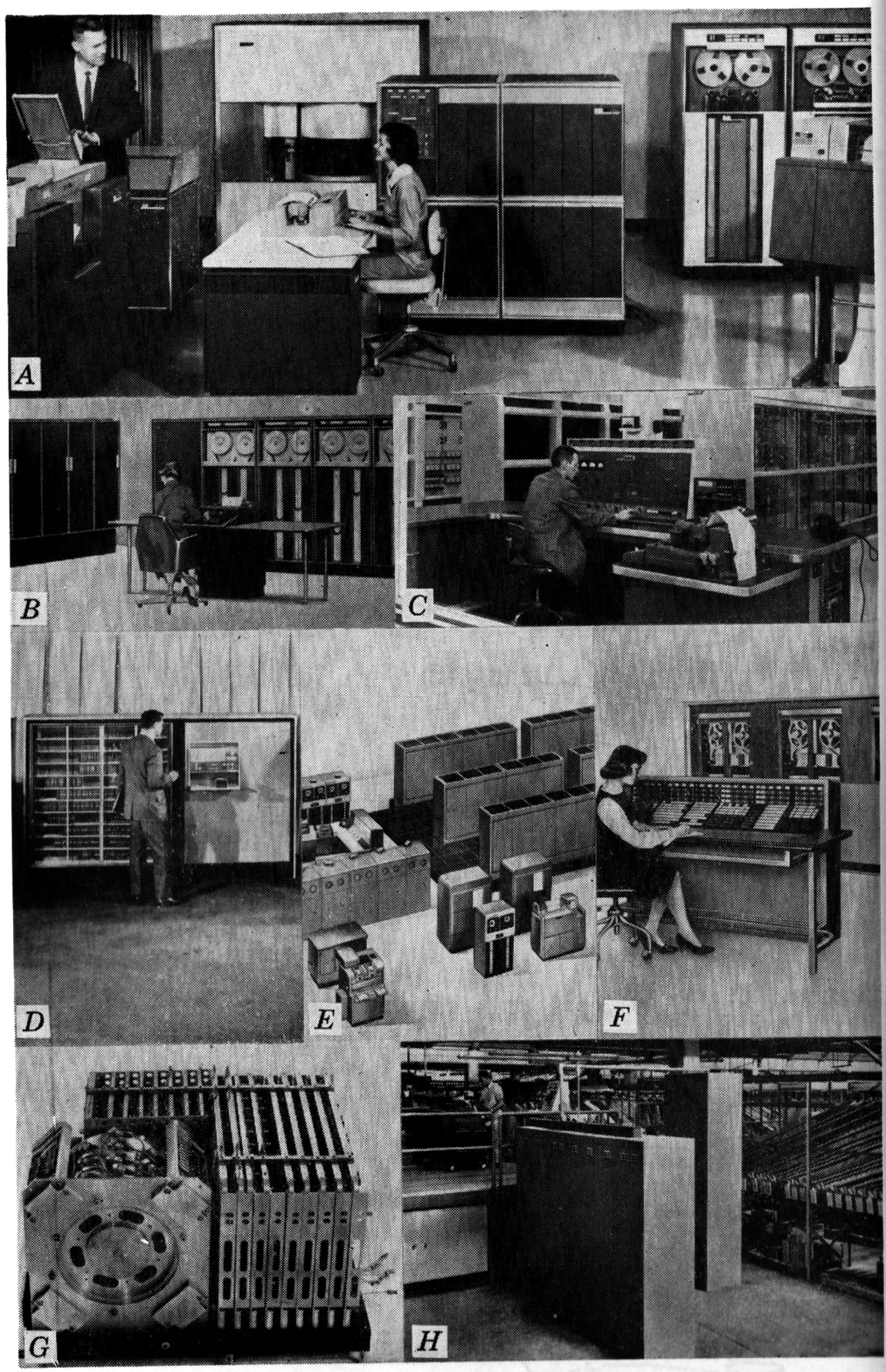


FIG. 1-2. Present-day computers. (*A*) IBM RAMAC 1401 data-processing system; (*B*) Honeywell 800; (*C*) Datamatic 1000; (*D*) UNIVAC solid-state computer; (*E*) UNIVAC 1105; (*F*) RCA 501; (*G*) air-borne computer; (*H*) RVS-Mark I (an automatic order-filling system). [*Courtesy* (*A*) *International Business Machines Corp.*; (*B, C*) *Minneapolis-Honeywell*; (*D, E*) *Remington Rand*; (*F*) *Radio Corporation of America*; (*G*) *Librascope, Inc.*; (*H*) *Industrial Electronics Engineers, Inc.*]

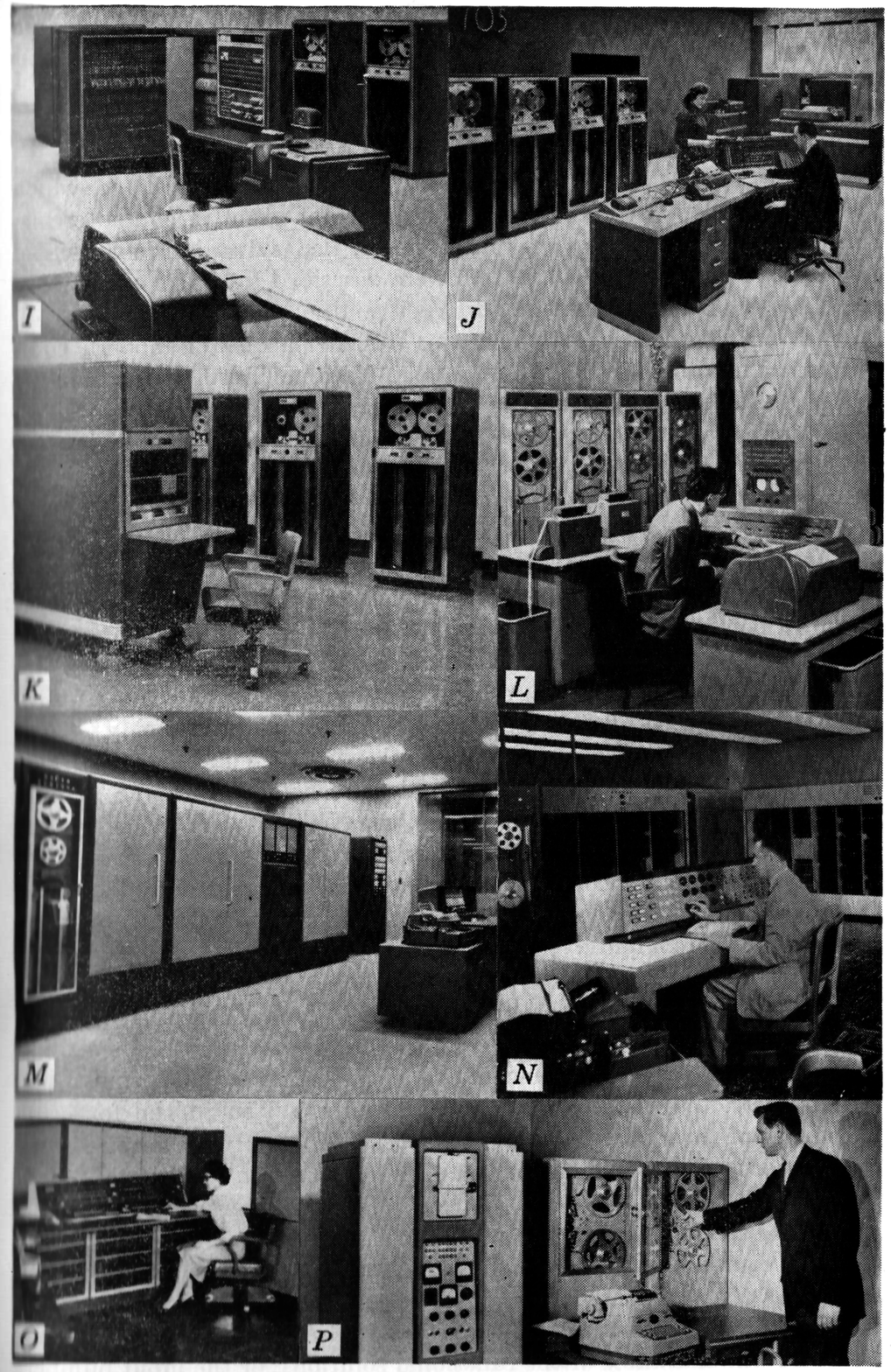


FIG. 1-2. Present-day computers (*continued*). (*I*) IBM 709; (*J*) IBM 705; (*K*) IBM 650; (*L*) Perseus G/7578/1; (*M*) Burroughs 205; (*N*) NCR 304; (*O*) TRANSAC-S-2000; (*P*) Bendix G-15. [*Courtesy* (*I, J, K*) *International Business Machines Corp.*; (*L*) *Ferranti, Ltd.*; (*M*) *Burroughs Corp.*; (*N*) *National Cash Register Co.*; (*O*) *Philco Corp.*; (*P*) *Bendix Computer Division.*]

tronic tubes. But these computers did not yet embody all the basic principles of modern computers.

The precursor stage was followed by the developmental stage, which lasted from 1946 to about 1952. An important event in computer history was a course given at the University of Pennsylvania Moore School in July and August of 1946. Here were described the basic concepts of the EDVAC, the design of which was being initiated at the Moore School. The EDVAC encompassed the major design features of modern computers. The design of the SEAC at the National Bureau of Standards and the EDSAC at Cambridge University, both of which computers were successfully operated even before the EDVAC, can be traced directly to that course. The IAS computer built at the Institute for Advanced Study can also be traced to the Moore School, as can probably many of the features of the Whirlwind computer built at MIT. But the SEAC, built from 1948 to 1950, was the first modern electronic computer in this country. Other pioneering computer efforts were successfully completed a few years later, and the fourth, or commercial, stage of computer history was initiated (see Figs. 1-1 and 1-2).

The names of S. Williams and G. Stibitz of the Bell Laboratories; H. Aikin of Harvard University; J. Eckert and J. Mauchly of the Moore School, University of Pennsylvania; M. V. Wilkes of Cambridge University, England; J. Curtiss, S. N. Alexander, R. Slutz, and S. Lubkin of the National Bureau of Standards; J. von Neumann, H. Goldstine, and J. Bigilo of the Institute for Advanced Study; J. Forrester of the Massachusetts Institute of Technology; N. Rochester and J. A. Haddad of the International Business Machines Corp., among many others, are outstanding in the computer field. In a sense it is unfair to mention only these names, for there are so many who have contributed their genius to the development of modern computers. In fact, it is probably one of the unresolved tragedies of modern technology that the results of individual genius are not long remembered as associated with that individual but are left to blend unidentified and unrecognized in the forgotten chronicles of these rapidly changing times. To do true justice to these brilliant scientists would require extensive research in itself.

The Role of the Programmer. In general, a computer can be conceived as a numerical-transformation machine. Numbers are the *inputs* to it, and the computer transforms these numbers into new numbers, which appear as the *outputs* (see Fig. 1-3). For instance, the input numbers may be the initial conditions of a differential equation, and the output numbers will be a table of the functional solutions; or the input numbers may be raw experimental data, and the output numbers will be the parameters determined by the experiment; and so forth. However, the computer must be directed to perform the required computations, and

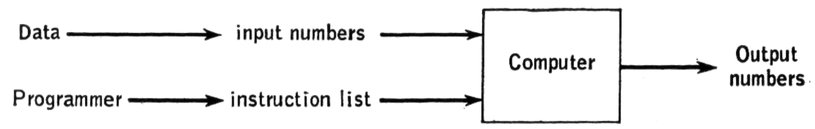


FIG. 1-3. Data flow to computer.

hence, in addition to the input numerical data, there must also be input *instructions*, which are coded in the form of numbers. Such a coded list of instructions is called a *program*, or code; the preparation of this program or code is the responsibility of the computer *programmer*. The purpose of this book is to describe the concepts, methods, and techniques involved in the preparation of a program, or instruction list, for a high-speed electronic digital computer. The role of the computer programmer is to utilize these methods and techniques to translate the verbal, mathematical, or other description of a computational problem into a list of instructions that will direct the computer to perform successfully the required computations.

A word should be said about the kinds of electronic computers. There are two main kinds, called *analog* and *digital*. In digital computers the numbers (digits) themselves are handled explicitly by the computer; an analog computer deals instead with a representation of the numbers, for instance, voltage, length, etc. In this book we are concerned with programming for digital computers only. These can be classified as *general-purpose* and *special-purpose* computers. Most of this book, however, with the exception of Chap. 4, is concerned with general-purpose computers, because a knowledge of these computers is a prerequisite for an understanding of the special-purpose computers. In Chap. 4 we shall consider some of the problems that arise in programming for special-purpose computers.

Chapter Outline. In the present chapter, we shall first consider applications of computers and then proceed with the principles of digital-computer programming. We hope that by presenting brief but relatively detailed descriptions of a few specific applications we may impart to the reader a better insight into how digital computers are used.

For the purposes of this chapter we have arbitrarily classified computer applications into four categories; many applications can fall into more than one of our categories. *First*, we shall consider *numerical solutions to equations.* This represents an important class of applications, and undoubtedly it is these applications which makers of the first computers had in mind. The *second* category is *process control:* here the need for digital computers with specialized capabilities becomes apparent. It is probably not an exaggeration to say that the *third* category actually came into being because of the great capabilities of the computers: this is

simulation. Before the advent of computers, detailed simulations were rarely even discussed because they were clearly not feasible to perform. However, with the use of high-speed computers, simulations have become practical, opening new fields of scientific research. Many of the present-day computer applications can probably be classed in the *fourth* category, *data processing:* Business accounting, statistical reductions, information retrieval, and many other important but routine procedures can be handled by computers.



New York Times.

LATE CITY EDITION
U. S. Weather Bureau Report (Page 58): forecasts: Fair and warmer today. Increasing cloudiness and milder tomorrow. Temp. range: 46—25. Yesterday: 39.6—27.8.

NEW YORK, THURSDAY, MARCH 19, 1959. 10 cents beyond 50-mile zone from New York City except on Long Island. Higher in air delivery cities. FIVE CENTS

U. S. ATOM BLASTS 300 MILES UP MAR RADAR, SNAG MISSILE PLAN; CALLED 'GREATEST EXPERIMENT'

BOARD BARS USE OF RESERVES TO HELP BUDGET

roposal Rejected
ts and Additional
Being Studied

UL CROWELL
of reserve funds to
e the city's 1959-60
get was unanimously
Mayor Wagner and
of Estimate yester-

ion was reached at
nute closed session
of Budget Director
D. Beame in the
uilding.
decision the Mayor
olleagues in effect
e the alternatives of
problem of budget-
imposing more nui-
denying or restrict-
lated pay rises for
es, curtailing city
by combining all or
se steps.
estion that the city
reserves was made
by Governor Rocke-
e Republican leaders
islature. They esti-
this would make
available for 1959-
purposes.

President Signs Bill For Hawaii's Entry

By C. P. TRUSSELL
Special to The New York Times.

WASHINGTON, March 18—President Eisenhower signed today the legislation to make the Territory of Hawaii the fiftieth state.

"It has given me great satisfaction to sign the act providing for the admission of Hawaii into the Union. Since my inauguration in 1953 I have consistently urged that this legislation be enacted, so the action of the Congress so early in this session is most gratifying," he said.

The Senate approved the enabling act by a vote of 76-15 on March 11. The House of Representatives passed it the next day by 323—89.

The President noted that the citizens of Hawaii would soon vote to decide whether their islands shall attain full statehood. A favorable decision of perhaps eight or nine

RADIATION SPREAD

Test Created Curtain Around the Earth For a Short Time

By WALTER SULLIVAN

Last September the United States drew a thin curtain of radiation around the earth. For a period whose duration is still secret it enveloped almost the entire inhabited portion of the globe.

Known as Project Argus, the feat was regarded by some of its leading participants as the greatest scientific experiment of all time. Thousands of men were involved in the world-girdling observations.

By prearrangement, rockets soared into space from the United States and abroad to monitor the effects. An American satellite—Explorer IV—

The New York Times March 19, 1959

BLASTS IN SPACE: Lines of force within earth's magnetic field determined shape of radiation shell produced by Project Argus. Explosions over Johnston Island (1) in August fired electrons along the magnetic lines to produce aurora at Samoa, but the effect did not envelop the earth. Argus shots (2) in September created an arc of fast-moving electrons that produced auroras over North and South Atlantic (3). Effect drifted east (arrow, 4) enveloping the globe within an hour. The drawing is schematic, and does not show exact site of Argus blasts.

3 DEVICES FIRED

Explorer Also Helped Gain Knowledge of Magnetic Field

By HANSON W. BALDWIN

Secret nuclear test detonations at more than 300 miles above the earth were conducted by the United States early last September. The tests, which have not previously been revealed and are believed to have been undetected by other countries, resulted in data of great military and scientific importance.

These data, though incomplete and controversial, indicate possible widespread interference with radar and radio by high-altitude nuclear detonations. In a military sense the findings have great potential significance to the development of an

FIG. 1-4. Announcement of Argus shots.

1-2. Numerical Solution to Equations

Motion of Charged Particles in the Earth's Magnetic Field. Project Argus was a major scientific and military experiment conducted by the United States (Fig. 1-4). Rockets carried a small atomic bomb more than 300 miles above the earth, at this altitude the bomb was exploded, and a resulting thin layer of radiation spread quickly round the globe. Nuclear and atomic particles were propelled by the exploding bomb and streaked through the vacuum of the universe under forces almost entirely due to the earth's magnetic field. Some particles moved toward the

earth's poles and caused artificial auroras (northern and southern lights). In order to measure the characteristics of this radiation layer, an earth satellite and other rockets were launched with paths passing through the layer (see Fig. 1-5). Instruments in the satellite and rockets transmitted measurements back to earth. Scientifically the shots rank among mankind's foremost experiments. They showed that radiation introduced

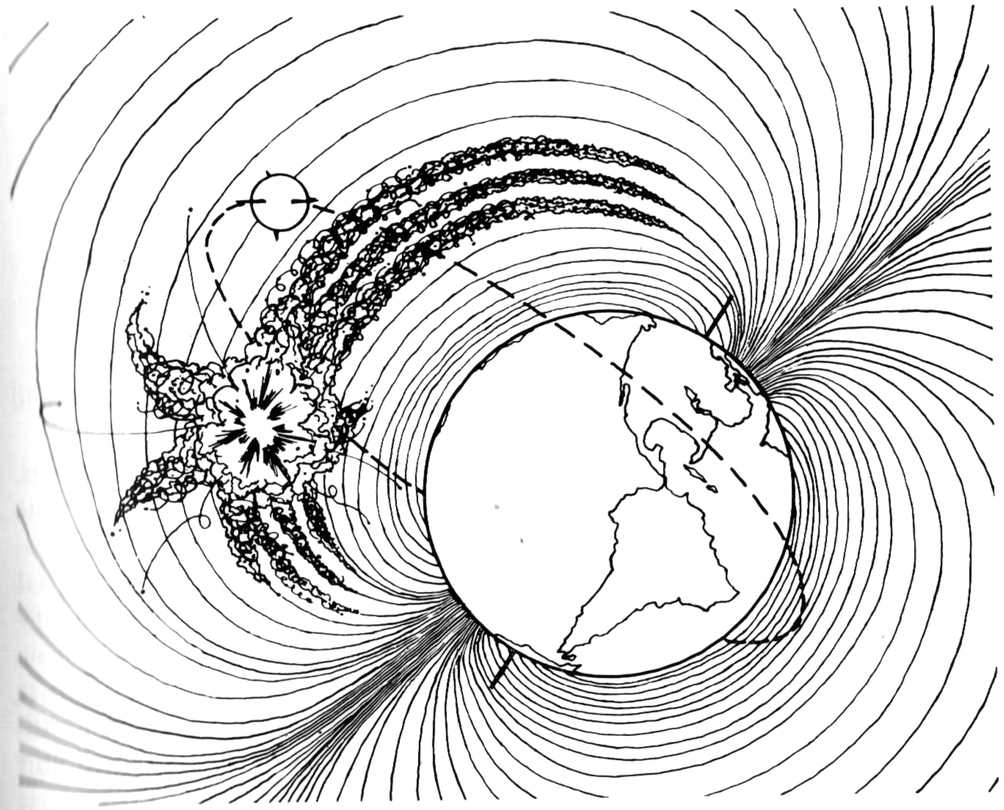


FIG. 1-5. Earth satellite measuring characteristics of radiation layer.

into space by man can have significant effects over those of natural radiation. Militarily the Argus shots indicated that the additional radiation added to the natural radiation interfered with man's electronic communications.

The Argus project tested theoretical calculations of what would take place when the small atomic bomb was exploded above the earth's atmosphere. This was a problem in the study of the motion of charged particles in a magnetic field (the earth's) and is in principle, except for the shape of the field, the same as the problems encountered in magnetically directing an electron beam in a television tube, or accelerating particles in a betatron, or circulating particles in a cyclotron, etc. If the earth's magnetic field is represented as that of a magnetic dipole, and if x, y, and z

are cartesian coordinates with the z axis directed along the earth's magnetic axis from south to north, then from electromagnetic field theory it can be shown that the trajectories of the charged particles are given by the solutions to the following differential equations (where the superscript dots denote differentiation with respect to time):

$$\ddot{x} = \frac{Mq}{mc}\{-(x^2 + y^2 + z^2)^{-5/2}[(2z^2 - x^2 - y^2)\dot{y} - 3yz\dot{z}]\}$$

$$\ddot{y} = \frac{Mq}{mc}\{-(x^2 + y^2 + z^2)^{-5/2}[(x^2 + y^2 - 2z^2)\dot{x} + 3xz\dot{z}]\}$$

$$\ddot{z} = \frac{Mq}{mc}\{+3z[k(x^2 + y^2 + z^2)^{-5/2} + (x^2 + y^2)(x^2 + y^2 + z^2)^{-4}]\}$$

where $k = (x\dot{y} - y\dot{x}) - (x^2 + y^2)(x^2 + y^2 + z^2)^{-3/2}$
M = earth's magnetic dipole strength
m = relativistic mass of particle
q = charge of particle
c = speed of light

The initial conditions are the initial coordinates (x_0,y_0,z_0) of the particle and the components $(\dot{x}_0,\dot{y}_0,\dot{z}_0)$ of its velocity just after the explosion.

These equations can be used to plot the paths of the charged particles, given their initial velocities and directions. The distribution of the initial velocities and directions can be calculated from nuclear physics; the trajectories of different kinds of particles can then be computed for many different initial conditions in the distribution. The complexity of these computations requires the use of a digital computer. The basic importance of such a calculation preliminary to the actual experiment is evident: by predicting the motion of the particles, proper and adequate plans can be made concerning the instrumentations of the satellite and the measurements to be made, the best geographical location of the experiment, the optimum altitude for exploding the bomb, and so forth.

Solutions to Equations in Biomedical Research. The partial differential equations that describe biological phenomena more often than not defy analytical solution. Attempts are usually made either to introduce simplifying assumptions or to isolate special aspects of the problem for investigation. Both these alternatives involve certain compromises on the part of a researcher. Such compromises can often be avoided, however, by using an electronic computer to solve the original partial differential equations numerically.

Computers are being used to assist in the complicated and extensive computations that are frequently involved in obtaining information about the precise atom structure or the over-all size and shape of crystallizable molecules, from X-ray diffraction patterns. Various organic molecules have been and are being analyzed by this method, including the well-

known cases of vitamin B_{12}, myoglobin, hemoglobin, lysozyme, and the hydroxyapatite of teeth and bone. These methods are also being applied in analyzing the structures of amino acids and peptides and the over-all shape of nucleic acids and other large molecules. For noncrystallizable substances, information about their structures can be obtained from scattered-light interference patterns, osmotic-pressure and viscosity measurements, and sedimentation rates. Here the computer could again significantly assist in the computations involved.

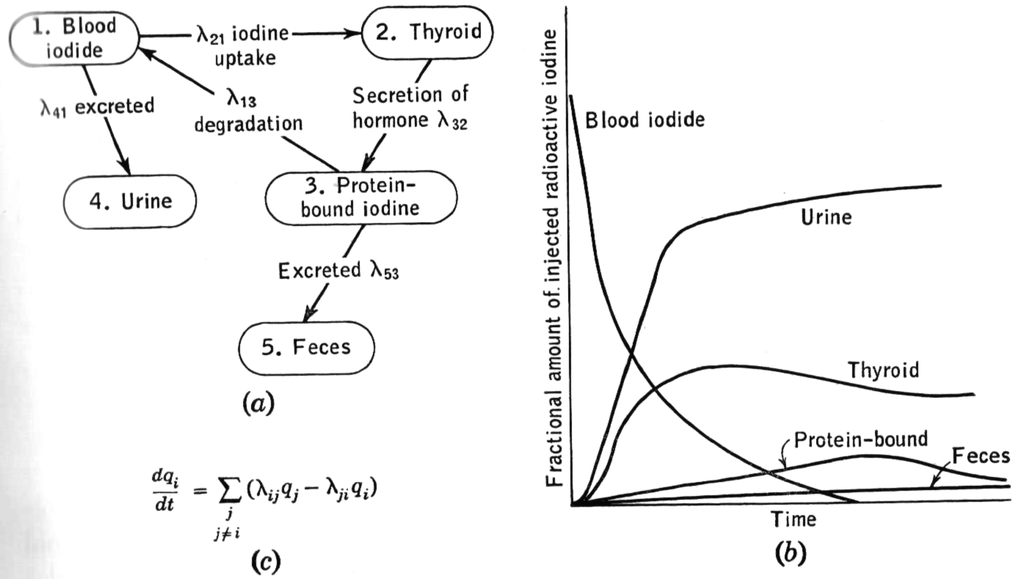


FIG. 1-6. Example of radioactive tracer study. (a) The five "compartments" of a thyroid tracer study; (b) typical measured levels of radioactive iodine in the compartments; (c) the set of five linear differential equations giving the dependence of the rate of change of amount q_i of radioactive iodine in compartment i, where λ_{ij} is the fractional turnover rate of radioactive material from compartment j to compartment i.

Another relatively active field in the use of computers is the study of differential equations relating to the mechanisms of biological systems. The famous Hodgkins-Huxley equations, and several variations thereof, have been analyzed with respect to the properties of excitable membranes. Besides the computations, well under way, concerning the nature of nerve-fiber conduction, other studies will apply these equations to muscle-fiber excitability. Other equations, relating to semipermeable membranes and the diffusion and motility of large molecules, could be solved numerically on the computer for complicated geometries and other specifications. This capability presents a further tool in the study of the nature of the physical phenomena involved in intracellular dynamics.

Complicated sets of interrelated differential rate equations can in many cases be solved on the computer. Such computations have been

applied to multiple-reaction-rate problems involved in the exploration of certain metabolic processes. Computations involving such sets of equations have been used to predict intermediate reactions from a knowledge of experimentally determined final reaction rates. Another important rate-equation problem to which computers have been applied is that of tracing through the various organs and systems of the body the time course of injected radioactive substances, for diagnosis of metabolic disturbances. For example, if radioactive iodine is used for a test of hyperthyroidism, the amount of radioactive substance would be measured as a function of the time in such "compartments" as the blood iodide, the protein-bound iodine, the thyroid gland, the feces, and the excreted urine. From these data the appropriate fractional-turnover rate equations are solved for the turnover-rate constants between compartments, these constants being the desired indices of disease (see Fig. 1-6). Facets of this problem are still being worked out, such as uniqueness of results, consistency of redundant information, statistical variation, and so forth. Radioactive-carbon-tagged chemicals can be used in this manner for pinpointing specific dysfunctions of the metabolic pathway, opening up new fields of research in computer-aided specific diagnostic analysis.

1-3. Process Control

Machine-tool Control. Significant improvements in many manufacturing processes can be obtained with the application of digital data-processing and control techniques. An example is digital machine-tool control, used to improve the over-all process of producing many different and complicated parts needed in our modern technology. Greater accuracy, reproducibility, and versatility are obtained as tooling costs and skilled-manpower requirements are reduced. It increases our capabilities to produce greater quantities and varieties of precision-machined parts, thus revolutionizing their potential use.

Consider a milling machine cutting a two-dimensional contour. The cutting tool is moved in the x and y directions by means of high-performance hydraulic servo drives. These servo drives are controlled by rapidly sequenced pulses that tell the servos precisely where to move the tool in order to obtain the contour finally desired. For instance, each time the x-direction control receives a pulse, it might advance the tool 0.0002 in. in the x direction; if we assume that there can be up to 20,000 such pulses per second, the tool can be made to advance in the x direction a maximum of $0.0002 \times 20{,}000 = 4$ ips $= 20$ fpm. By varying the number of pulses per second, up to the maximum, the *speed* with which the tool moves in the x direction can be precisely controlled. The control of the y-direction motion of the tool is similar. Thus the tool can be directed to move round any desired contour. The precision of machining

the desired part is of course limited by the smallest increment of distance of the control.

The problem of digital machine-tool control reduces to that of transforming information on a blueprint design into an appropriately timed sequence of x and y input control pulses. The control pulses direct the center of the cutting tool, but the desired contour is formed by the edge of the cutter; hence the proper *cutter offset* must be calculated from the

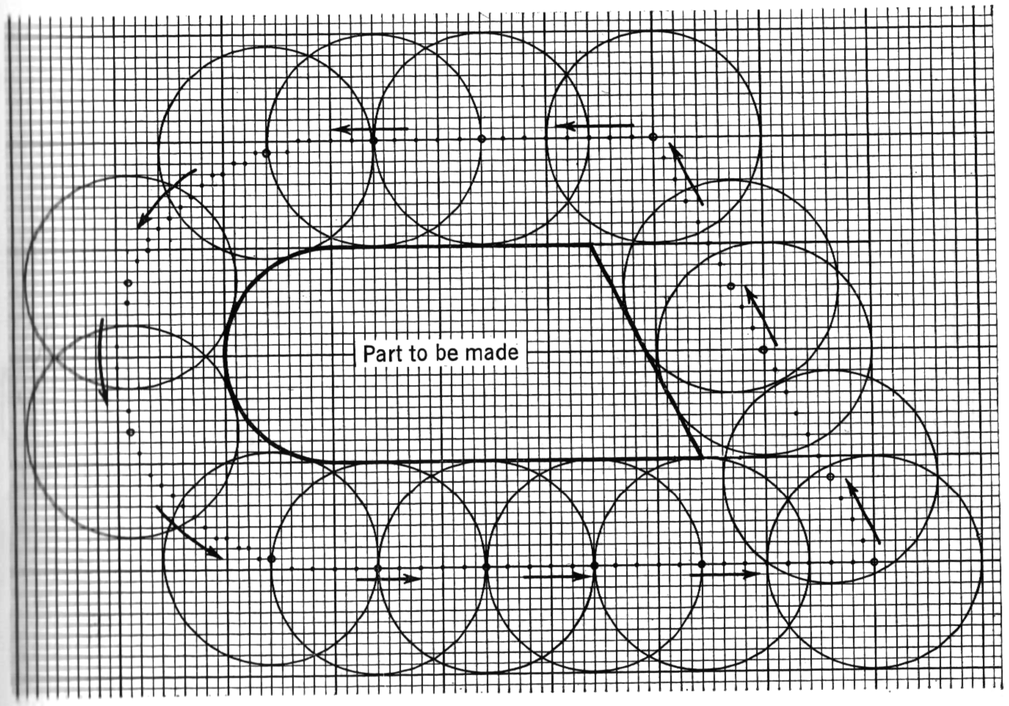


FIG. 1-7. Cutter offset path.

blueprint. See, for example, Fig. 1-7, where the circles indicate some of the successive positions of the cutter. If we assume in Fig. 1-7 that the graph lines are 0.0002 in. apart, then the dots represent the successive positions of the offset path of the cutter's center that should be directed by the control pulses. (For example, the diagonal cut would be made by two pulses in the y direction for each one pulse in the x direction.) Of course, in a realistic case the part would be many times the size illustrated, and hence the number of input pulses required would be very large. Thus it would be a difficult job to start from the blueprint and determine manually the detailed sequences of control pulses. The method of interpreting the blueprint design for the automatically controlled milling machine is divided into two steps. *First* the cutter's offset path is determined grossly in terms of the end points of straight-line path

segments, radii of circular contours, and so forth. This in itself can present difficult problems of curve fitting. The *second step* is to interpolate between these points and from the radii, etc., to determine finally the precise 0.0002-in. *input-pulse fine structure.*

Two computers are used for this process. The first is a general-purpose digital computer into which are put the specifications for the part contour taken from the blueprint. These might take the forms "diagonal at arctan 2," "straight 4 inches," "horizontal 3 inches," "radius 2 inches," etc. Or the input to the digital computer might be the blueprint itself,

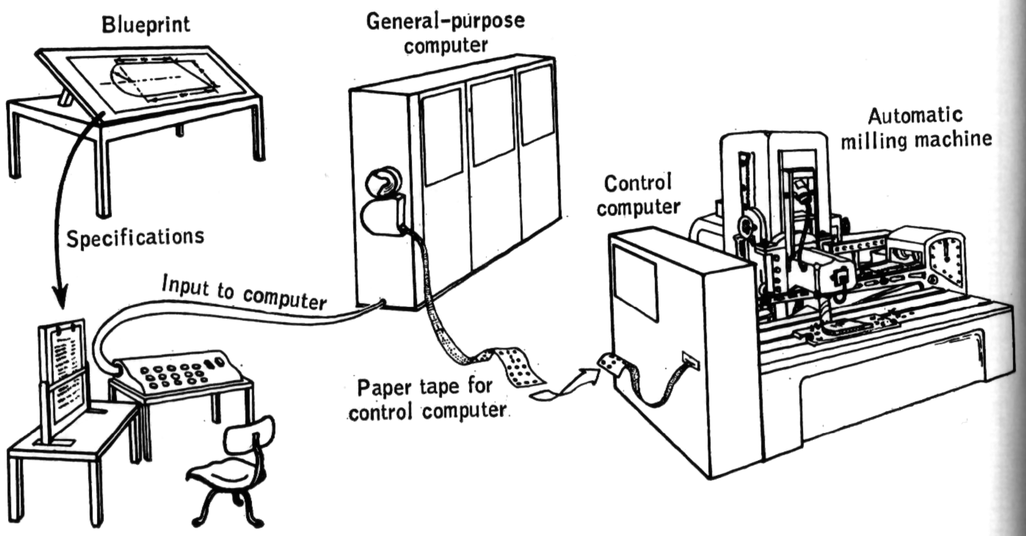


FIG. 1-8. Machine-tool control process.

put into the computer by means of a "picture-reading" machine. The computer is programmed to compute from such input data the gross data of the offset cutter path. This is perhaps put out in the form of punched paper tape, like the rolls of a player piano, or in the form of a deck of punched cards. The second computer is used to control the milling machine directly. The punched paper tape or card deck is put into this control computer, which interpolates, etc., to produce the fine-structure input pulses to the servo drives. The entire process is illustrated in Fig. 1-8.

Control of Chemical Plant. In many factories the processing of raw materials frequently involves sequences of decisions for which all alternative possibilities are known. By means of digital computers the entire processing can be made completely automatic. Control of this kind has the advantages that the decision processes themselves become more reliable when performed electronically and that the design of more complicated processes becomes feasible. As a specific example, consider a

simplified version of a chemical factory that produces Al_2O_3 (see Fig. 1-9). Table 4-2† summarizes the conditions which initiate various actions in the plant. The column entitled "Sensed-signal conditions" represents the list of conditions requiring some operation to be performed. The column entitled "Corresponding signals to be generated" indicates the actions to be taken.

Input signals to the control computer are transmitted by wires from critical places in the factory (see Fig. 1-9). These signals, defined in Table 4-1, consist essentially of two-valued (on or off) test results which in combination tell the computer the *present state of the factory.* For example, signal $L(1)$ will be a pulse when the mixture in mixing tank 1 is at or above a predetermined level and no pulse if the mixture is below this level. Each signal is initiated by a *transducer,* a device that converts mechanical information into electrical information or vice versa. The transducers for our example convert fluid levels in tanks, temperatures, chemical concentrations, volumes of precipitate in a centrifuge, pressures, and so forth, into the electrical pulses that are the inputs to the computer. The outputs from the control computer are electrical signals supplied to transducers (servomechanisms) that open and close valves or hatches and otherwise operate to change the state of the factory. Table 4-2 shows under what conditions such changes of state will be initiated.

1-4. Simulations

Industrial Dynamics. The term industrial dynamics refers to the processes and factors influencing the economic development of an industrial corporation considered as an interacting part of a free competitive economy. The study of industrial dynamics involves so many factors that no complete analytical mathematical analysis is yet possible. Many universities in the United States, in cooperation with many industrial corporations, are attempting to simulate aspects of the industrial-dynamics problem. Representatives of management take part in such simulations, making decisions; the digital computer then computes the effects of these decisions and produces a report to the management representatives, who make new decisions, etc. (see Fig. 1-10). One advantage of such simulations is that they aid in training management personnel by presenting them with many and different industrial circumstances analogous to real situations. A second advantage is that the simulations aid the study of industrial dynamics by bringing before the scientists involved situations not encountered through purely academic

† The control of this particular process will be considered in greater detail in Sec. 4-3; it suffices here to outline briefly the kind of input information supplied to the computer and the way the computer then controls the factory. (See pp. 132 to 137.)

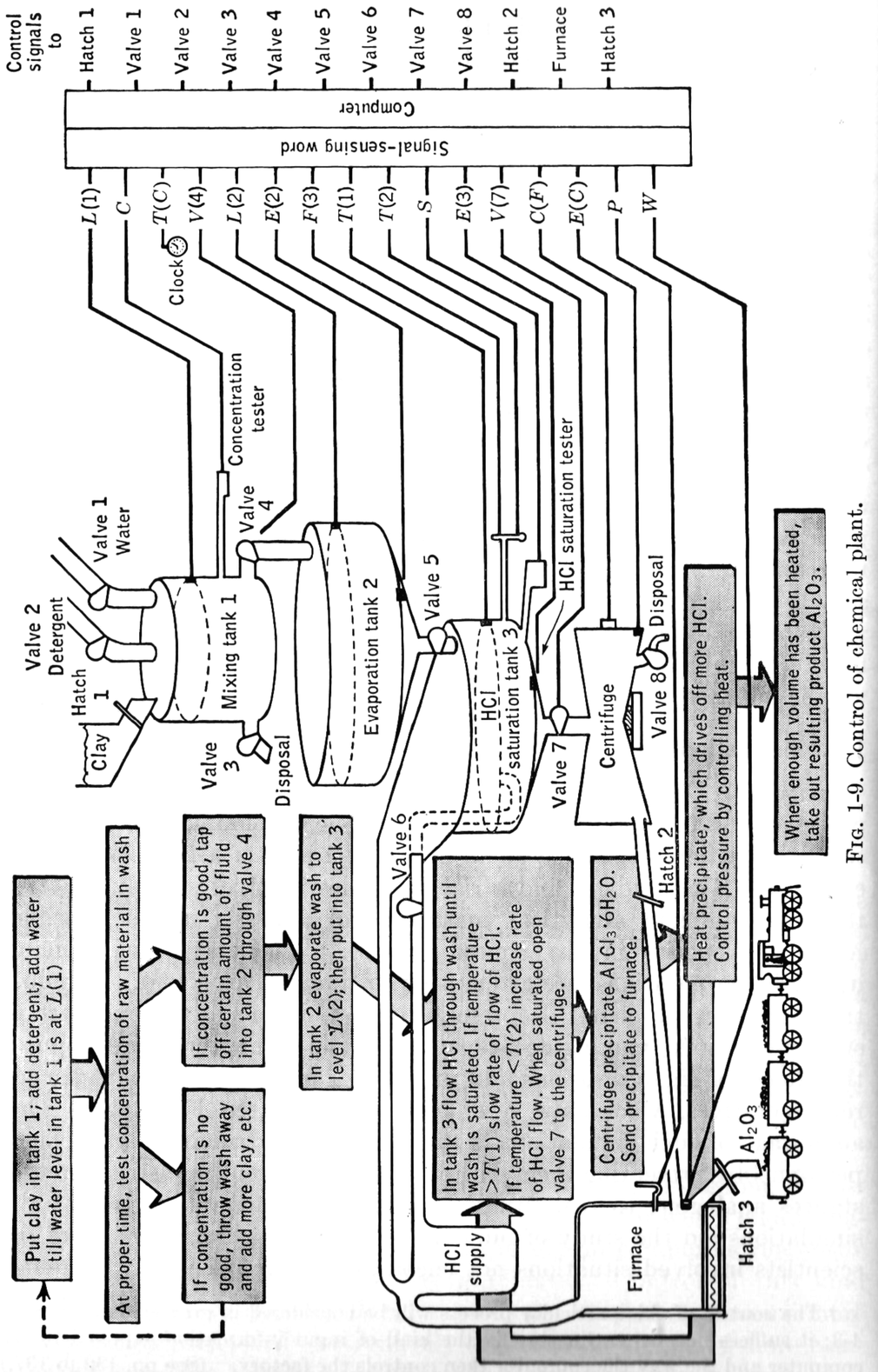


FIG. 1-9. Control of chemical plant.

study. And, third, important generalizations can be deduced from the progress and results of the simulations.

As an illustration, consider a multifirm one-product competitive industry. The simulation is designed in such a way that the results of 5 years of operations can be simulated in 1 day. Each firm in the simulation is represented by a team of executives, who must make the following set of "management" decisions:

1. Price of product
2. Producing volume
3. Advertising expenditures
4. Research and development expenditures
5. Amount of new investment in plant and equipment
6. Dividends to be paid, etc.

One set of such decisions is made by each team firm for each quarter of a year. The computer then computes the interactions and effects of these decisions on the market; i.e., it simulates what may happen in reality and produces for each team a report giving the following:

1. Sales volume
2. Per cent of total products sales handled by this firm
3. Production capacity of this firm for the next quarter
4. Statement of profit and loss
5. Current inventory quantities
6. Statement of receipts and disbursements
7. Statement of financial conditions, etc.

On the basis of study of its report, each team makes decisions on how the firm will operate in the next quarter. These decisions are fed into the computer, and the simulations progress in this way from quarter to quarter.

The computer processes the decisions made by the teams (firms) and produces reports by means of mathematical models. The model is essentially a set of hypothetical equations relating the input information to the desired output information. For example, increased advertising bears a functional relation to increased sales, but this in turn depends on the advertising decisions of competitive firms. Increased research and development can produce an improved product which may give one firm some advantage. Price rises of course offset sales, but advertising can to a certain extent offset an increase in price. Obviously the gross profit is the product of sales volume and price per unit product, minus administrative costs, overhead, manufacturing costs, advertising, research and development, etc. The mathematical model embodying such considerations can become very complicated, depending on the level at which the simulation is intended. Evidently a great deal of study and research must be carried out in connection with the derivation of

the specific equations that enter into the computer simulation. For a simulation to be at all realistic, the complications will be so numerous as to be impossible to perform without digital computers.

Man-Machine Simulation System. Man-machine simulation is now recognized as an essential tool for the development of complex automatic systems that involve both electronic and manual controls. Such is the case, for example, in air-traffic control, ground control of interceptors, ground-controlled approach and landing, weapon assignment, or missile guidance systems. The interactions between the electronic control system being designed and the team of human operators can be studied *before* the system is actually constructed. This eliminates the only other alternative, the frequently prohibitively expensive process of actually building the proposed system and carrying out an extensive program of tests and modifications.

Figure 1-11 represents a simulation system that includes both a digital and an analog computer, linked with display and control equipment which permit human beings to be coupled into the system. The analog-computer equipment is used for such tasks as accepting continuous control information and solving complex dynamic equations in real time, for which they are suited. The digital computer is used for controlling the experiment, for sequencing the stages depending on the outcome of each stage, for generating simulated data, for calculating with high precision where necessary, for handling variables with large dynamic ranges, for making logical decisions, and for statistically analyzing experimental results.

An example of a simulation is a simplified ground-controlled interceptor system (Fig. 1-11). The digital computer simulates a situation in which a large number of aircraft are observed by the search radar. A console displays the locations of these aircraft on a cathode-ray tube for the ground crew; with each aircraft is also displayed an associated identification (i.e., friendly, enemy, unknown, interceptor, bomber, commercial, etc.). The ground crew manually assigns targets to the interceptors; the targets are automatically tracked by the digital computer. A model of an interceptor cockpit fully equipped with instruments and pilot flying controls enables a human operator to imitate the actions of the interceptor pilot in flight. The pilot of the interceptor receives his flight-control information either directly from the digital computer or from the ground-crew commands. The pilot's control movements are interpreted by the analog computer, which then simulates the dynamics of the interceptor, operating the instruments in the cockpit. The velocity components of the interceptor are sent to the digital computer, which computes the path of the interceptor. The digital computer also displays the computed positions of both interceptor and target on a

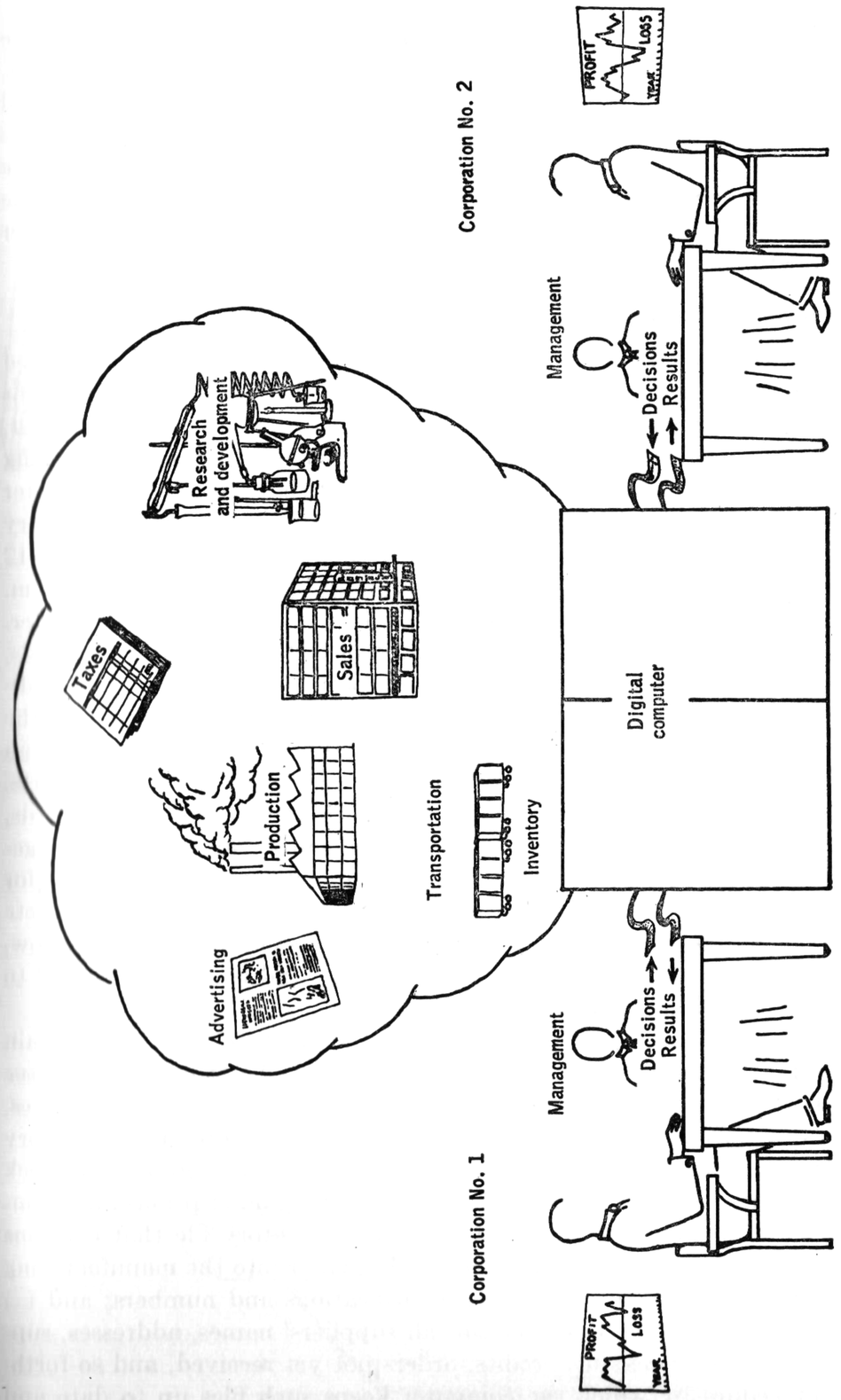


FIG. 1-10. Business simulation.

monitor scope for the ground crew. It is this display which gives the ground crew its information for the simulated data link with the pilot. The research control console enables a research team to set up the initial conditions, vary the target aircraft maneuvers, and determine the interceptor closure tactics. Records are kept of information generated by the interceptor pilot, analog computer, and digital computer during the experiment. These can then be analyzed on the digital computer after the experimental simulation has been completed.

1-5. Data Processing

Inventory Control. The annual industrial inventory of the United States has been estimated at 30 billion dollars; the cost of supporting this inventory is probably about 3 billion dollars annually in interest lost, tied-up money, depreciation, obsolescence, spoilage, storage, handling charges, and so forth. It has been estimated that the use of computer control and information-processing methods could reduce this inventory and its associated cost of support by more than 25 per cent. Figure 1-12 indicates how a computer could be used in an inventory-control system. Here the inventory files are electronically recorded in the computer. Inputs to the computer are items initiated outside the inventory records, such as receipts, issues, orders, shipping reports, etc. These inputs indicate the alterations in the inventory records that must be made by the computer. The outputs of the computer are items that arise from an examination of the inventory records, such as the need for more parts, orders to restock, bills or invoices for items shipped, accounting records, and other data about the current inventory records that may aid management in planning, etc. The analysis of an inventory-control problem for the use of a digital computer includes the preparation of a complete report on all the processing requirements, process charts of the data flow, and definitions of the controls. These reports and charts correspond to the equations describing a scientific problem.

Consider, for example, a concern that manufactures and sells certain products, and suppose that the company keeps the following files (see Fig. 1-12): (*a*) a customer master file that contains all customers' names, addresses, purchases, balances due, credits, etc.; (*b*) a products inventory master file that contains the kind and number of each product in stock as well as its location, price, and other identifying descriptions and numbers; (*c*) a factory parts and raw-materials inventory file that contains a list of all parts and materials stocked that enter into the manufacturing process, together with descriptive identifications and numbers; and (*d*) a suppliers master file that contains all suppliers' names, addresses, supply records, balances due, credits, orders not yet received, and so forth. The procedure by which the computer keeps such files up to date and

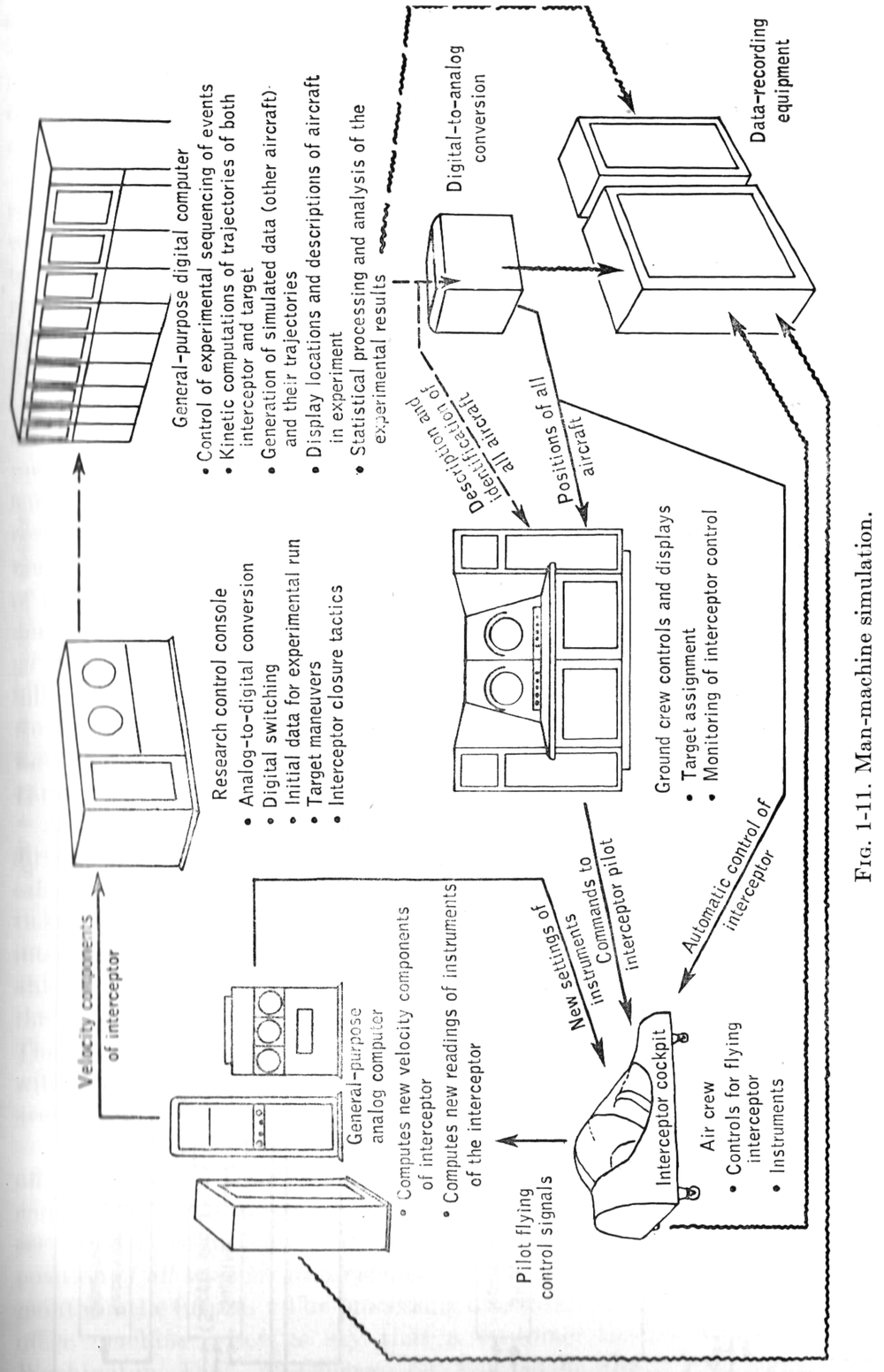


FIG. 1-11. Man-machine simulation.

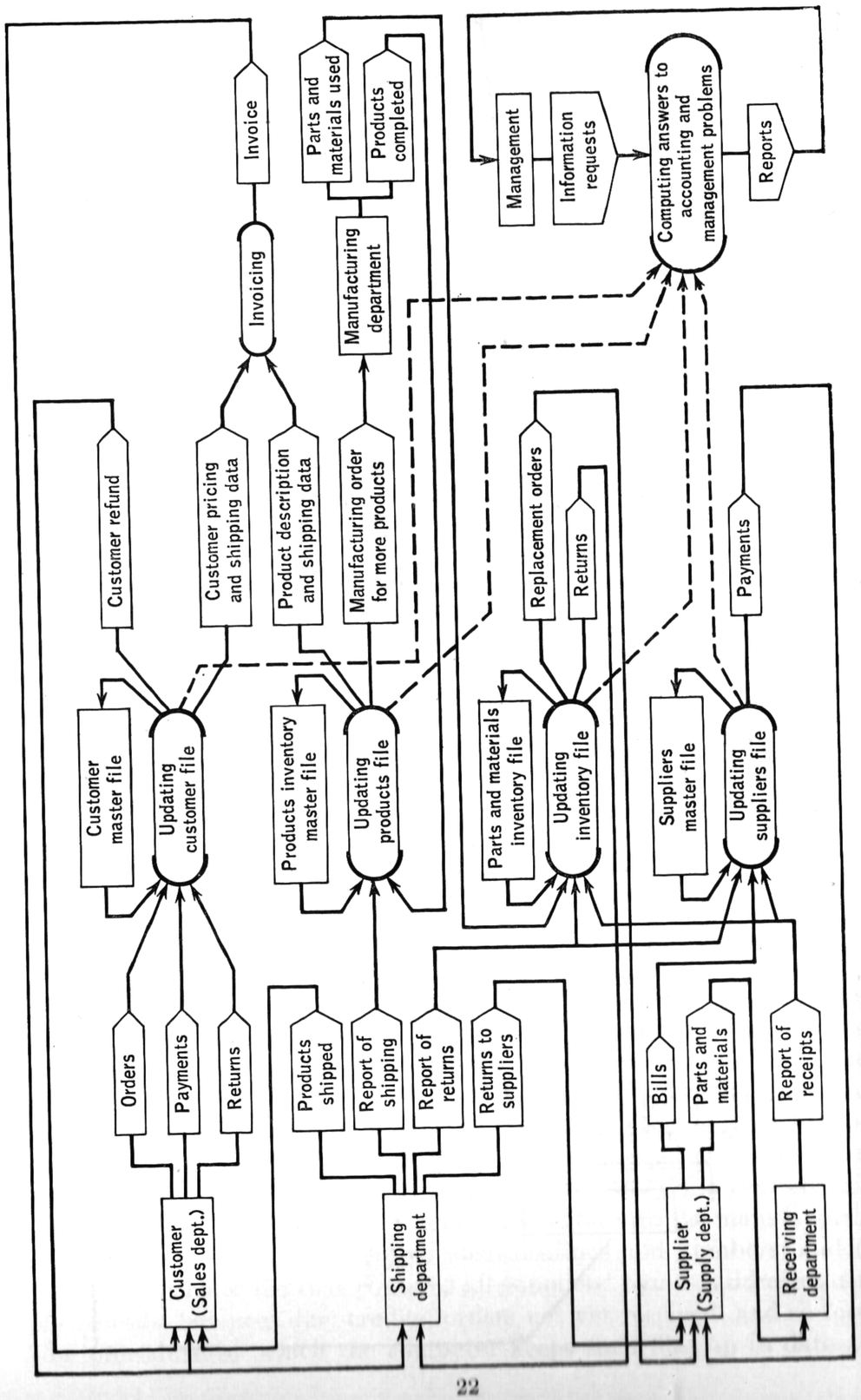


Fig. 1-12. Inventory-control system.

generates information from them will now be described; the reader should keep in mind that our intention is to give a gross idea of the problems involved, and we have hence oversimplified the circumstances. More realistically, these problems are characterized by huge masses of detailed considerations.

The customer file would be updated each time an order, payment, or return was received from the customer. At the same time the computer would produce the necessary pricing and shipping information (i.e., customer discount, location, etc.) if an order was made, or the customer's refund order if a return was made. The products inventory file would be updated each time a report was received of shipping or completing a product. If a product was to be shipped, the computer would produce the product description number and warehouse location, and it would also produce an order for more products to be manufactured if the shipment reduced the inventory below a predetermined level. The computer can also produce the invoice for a sale from the proper information received from the files. The file of parts and raw materials used in the manufacturing processes would be updated each time a report was made of parts and materials used, returned, or received. The computer would determine what replacement orders and returns should be made as well as update the inventory. The supplier file would be updated each time a bill was received or report of return or receipt was made. The computer would produce the payment orders to the supplier. Finally the computer would produce reports requested by management, on the basis of the various file records.

Airline Reservation System. In the reservation system of a nationwide airline, every minute hundreds of inquiries about seats available, actual sales, cancellations, and unsatisfied demands must be processed. Each ticket office in the United States of a particular airline must be able to interrogate the same centrally located computer concerning seats available on particular flights and to make reservations on particular flights; the processing of each query must be completed within a few seconds. The main advantages of such a system are that the customer is provided with an immediate reservation service and the airline is provided with an accurate inventory system almost devoid of errors.

A simplified reservation system is shown in Fig. 1-13. Each ticket office contains a reservation interrogation and sales machine, which is connected by transmission lines to a central service area. At the central service area, records are electronically stored on the availability and disposition *of all seats on all airplanes* to be flown by the airline, for about a month in the future. The processing of a reservation starts at the ticket-office machine. Let us say that a customer desires two seats from Washington, D.C., to Chicago on Apr. 30 on the 3 P.M. flight. The

sales clerk observes that Flight 83 leaves Washington, D.C., at 3 P.M. and goes to Seattle via Chicago, and chooses the card for Flight 83 from a file box. Each flight card contains information about two flights, one on each of the long edges. Along the edge are printed the scheduled stops, leaving a margin which is punched with a code representing the flight number. The clerk puts the punched margin of the card into a slot in the ticket-office machine, then presses buttons indicating the number of seats (two) and the date (Apr. 30) desired. The names of the

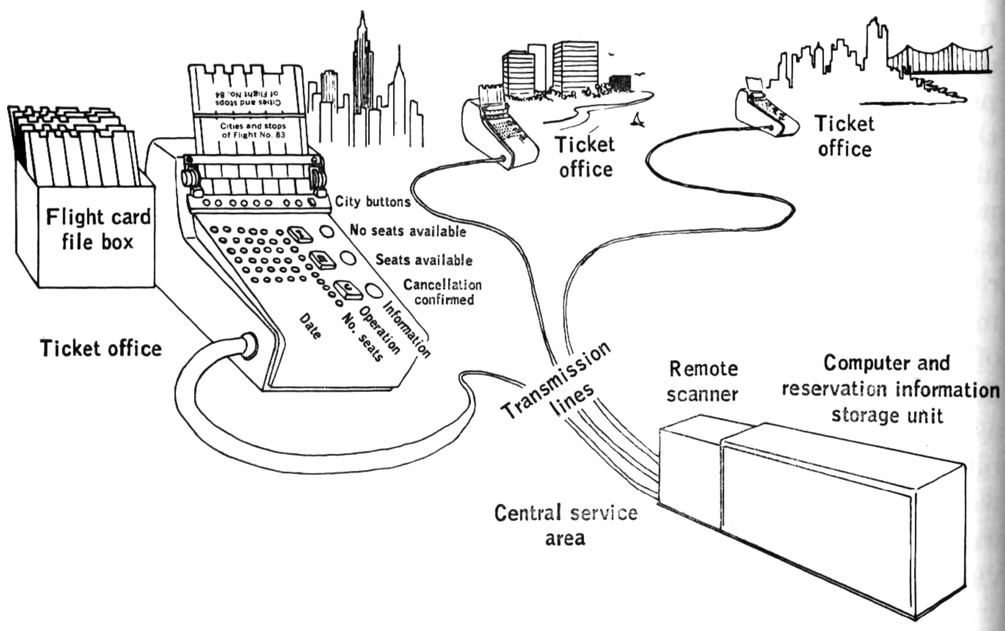


FIG. 1-13. Airline reservation system.

cities on the card appear opposite a row of buttons on the machine, and the clerk pushes the buttons next to the names of the desired cities, Washington and Chicago. Then he presses the *I* (interrogation) button. Meanwhile the remote scanner (see Fig. 1-13) is searching for a ticket-office machine with a card in it. When one is located, the remote scanner connects the ticket-office machine with the computer. The marginal punches on the card are sensed by the computer and tell it which flight is being interrogated; the date, number of seats, etc., are also transmitted to the computer. If two seats are available on Flight 83 for Apr. 30, the computer transmits the information to the ticket-office machine by illuminating the proper lights on it. If the customer desires to make the reservations, the *R* (reservation) button is pushed and the computer subtracts two from the number of seats available on Flight 83. If no seats were available, the no-seats-available light would have been illuminated. Similarly a cancellation can be made, and so forth. Of

course our illustration has been greatly simplified, but the essential operations performed by the computer and the problems involved have been demonstrated.

Aids to Medical Diagnosis. It has been recognized that electronic computers can aid certain aspects of medical diagnosis. For example, the computer can (1) produce a list of possible diagnoses, consistent with medical knowledge, for a given set of symptoms presented by a patient; (2) indicate further diagnostic tests which best differentiate between remaining diagnostic possibilities; (3) calculate the probabilities for the alternate diagnostic possibilities; and (4) enable a more precise statement and analysis of the value decisions which may be associated with treatment planning. Such computer applications must be based on extensive medical data; hence a further use of computers is (5) to compile statistics that (*a*) relate symptom-disease combinations and that (*b*) evaluate disease-treatment-prognosis results. These statistics are then used in future diagnosis, treatment, and prognosis determination. In addition, the computer can serve as an indispensable aid, under certain special circumstances: The computer (6) can enable criteria of a more quantitative nature to be utilized in evaluating the results of certain diagnostic procedures (such as electrocardiograms, electroencephalograms, etc.) and can perform the complicated calculations necessary for the proper interpretation of certain clinical measurements. Finally, the computer can significantly aid the important aspect of information retrieval in, for example, (7) the rapid retrieval of current information about new preventive measures, diagnostic techniques, and specific treatments, and (8) the accumulation and retrieval of desired aspects of a particular patient's total medical history (such as total radiation dosage received, previous allergic reactions, individual biochemical and physiologic norms and deviations, etc.).

Consider a modern high-speed electronic digital computer, and suppose that it has been programmed to aid medical diagnosis. Let us assume further that the physician can directly communicate with the computer by telephone, teletype, radio, etc. The value of such a computer interrogation arises from three factors: (1) the ability of the computer to formulate a treatment plan that will maximize the chance of curing the patient; (2) the ability to determine the minimum number of necessary medical laboratory tests or other diagnostic procedures for the particular patient; and (3) the ability to evaluate more accurately diagnostic-test results for a particular patient based upon his previously recorded health records.

A network of such computers could form a hypothetical health-computing system (see Fig. 1-14). Here each computer can communicate with individual physicians and hospitals within its area, receiving, transmitting, and computing medical information as required. The great sig-

nificance and importance of such a health-computer network cannot be overestimated as an aid to increasing individual good health and longevity and as a vast new source of medical information concerning mankind.

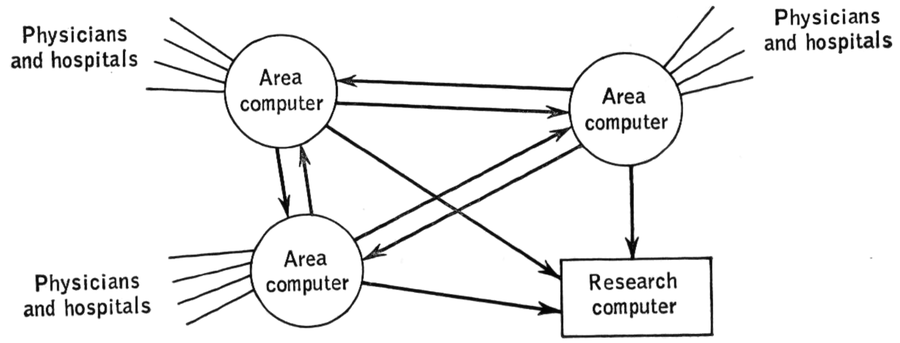


FIG. 1-14. Hypothetical health-computing system.

1-6. Block Diagram of a Computer

The purpose of this section is to acquaint the student briefly with the parts of a digital computer. To this end the block diagram of a computer will be developed in steps, from the simple to the more detailed.

The Computer. As was seen in the previous sections, a computer handles numbers; it takes numbers that are fed into it and operates on them to form new numbers—the output numbers. Hence the simplest block diagram of a computer is as shown in Fig. 1-15.

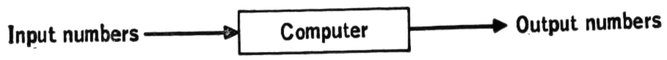


FIG. 1-15. Simplest block diagram.

The computer can perform many different kinds of *operations* on numbers. Most of these operations combine two numbers to produce a third number. The two numbers so combined are called the *arguments* of the operation, and the number produced is called the *result.* Many operations that a computer can perform are similar to the familiar operations used in hand computation, such as adding two numbers to form the sum, multiplying two numbers to form the product, etc. Also analogous to hand computing is the fact that the computer can perform *only one such operation at a time.*† Hence, when many operations are to be performed, *they must be performed in some appropriate sequence.*

The computer is told what operations to perform by means of *instructions.* An instruction not only contains information about an operation to be performed but also designates the two arguments (usually) for the desired operation. When the operation specified by an instruction has been completed by the computer, i.e., when the result has been obtained, the instruction is said to have been *executed.*

† Modern computers are being developed to perform several operations simultaneously.

For example, suppose that we wanted to compute the value of

$$y = a(b + c)$$

where a, b, and c represent some specific real numbers. A sequence of instructions that would tell a computer to form $a(b + c)$ is as follows:

	Operation	1st argument	2d argument	Result
Instruction 1	Add	b	c	$x = b + c$
Instruction 2	Multiply	a	x	$y = ax$

The computer would first execute instruction 1 and then instruction 2.

The Memory and Computing Unit. The computer itself is composed of two main parts, a *memory,* which stores numbers and instructions, and a *computing unit,* in which the actual computations take place (see Fig. 1-16). The computer's memory stores the numbers to be operated on;

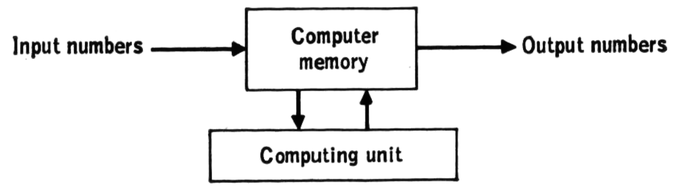


FIG. 1-16. Memory and computing unit.

it stores intermediate results that are generated during the course of a computation; and it stores the final results. For instance, the numbers represented by a, b, and c of our example are initially stored in the memory. After the first instruction is executed, the intermediate result x is stored in the memory. After the second instruction is executed, the final result y is stored in the memory.

The instructions themselves are also stored in the computer's memory. Each instruction is transmitted to the computing unit when it is to be executed. In our example, instruction 1 is first transmitted to the computing unit and executed, and then instruction 2 is transmitted to the computing unit and executed.

The reason for having an internal memory in the computer is that numbers and instructions can be transmitted between such a memory and the computing unit more rapidly than by other means. This enables

the computations to proceed more quickly. There are high-speed computers in which a number or instruction in the memory can be transmitted to the computing unit in as short a time as 1 μsec. On the other hand, arithmetic operations such as addition can be performed in less than 1 μsec. Hence, it is generally the speed of the memory that limits the speed of a computer.

It is seen from the above discussion that the computing unit has two functions: it must (1) obtain instructions from the memory and interpret them, as well as (2) perform the actual operations. Hence it is composed of two parts: the *control*, concerned with the former, and the *arithmetic unit*, concerned with the latter function (see Fig. 1-17).

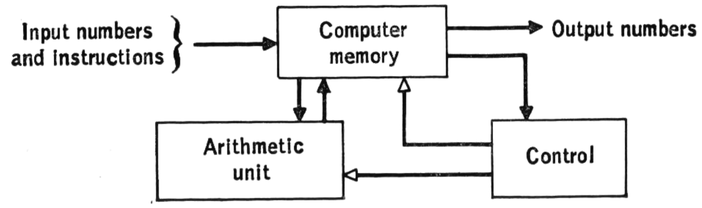


FIG. 1-17. Arithmetic unit and control. (Solid-headed arrows indicate information; hollow-headed arrows, control signals.)

Analogy between Hand Computation and the Functions Performed by the Parts of a Digital Computer. A general rule for evaluating the capabilities of a digital computer is this: If the steps in the solution of a problem can be broken down into sequences of unambiguous instructions that conceptually could be performed by a very patient secretary who has no knowledge of the subject matter but who has infinite perseverance and can follow instructions precisely, never making an error, then the problem can be solved on a digital computer. This rule is an oversimplification only in so far as the necessary computation time on the computer and the cost of this time are also restrictions. It is instructive to carry our analogy still further, describing the functions of the various parts of the computer (see Fig. 1-18). The computer memory is analogous to the worksheet and instruction list of the hand computation: the memory stores the initial data to be operated on, the intermediate and final results, and also the instructions themselves. The control interprets the instructions regarding the operation to be performed, the arguments to be operated upon, and the instruction to be chosen next; this is analogous to the computing secretary's mind. The actual operations indicated by the instructions are performed in the arithmetic unit, analogous to the secretary's desk calculator.

The Control. The control itself must perform two functions. It must (1) interpret the instruction; then, on the basis of this interpretation, it

must (2) tell the arithmetic unit what to do. The latter function is accomplished through the use of electronic control signals.

In accordance with these two functions, we can separate the part of

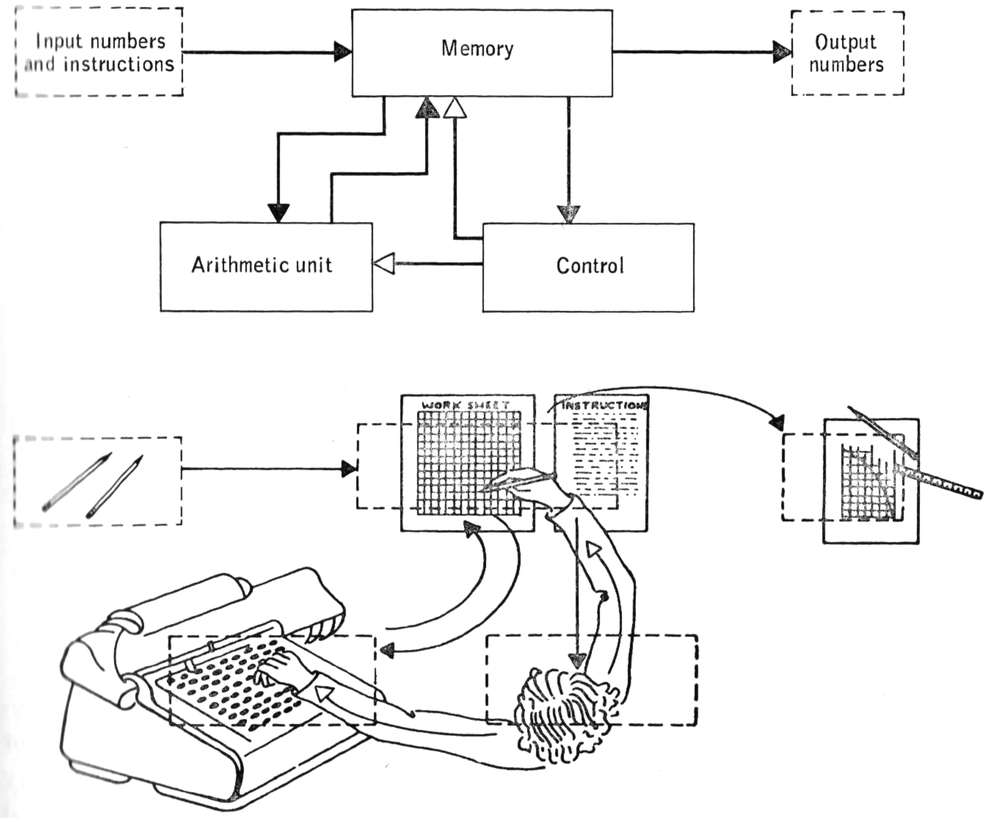


FIG. 1-18. Analogy between manual computing and an electronic digital computer. The arrows indicate the flow of information and control signals.

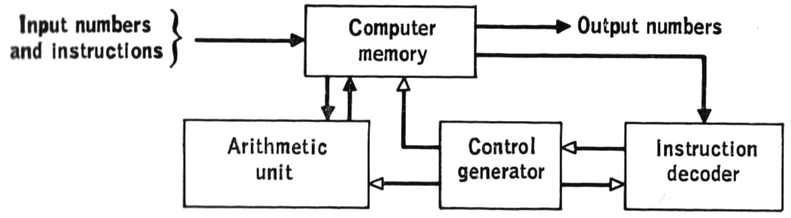


FIG. 1-19. Instruction decoder and control generator.

the control that interprets, or *decodes*, the instructions, called the *instruction decoder*, from the part that generates the control signals, called the *control generator* (see Fig. 1-19).

After an instruction has been transmitted to the instruction decoder, where it is interpreted, the control generator senses this interpretation

and then produces signals that tell the arithmetic unit which operation to perform. It also generates signals that choose the proper numbers (arguments) from the memory and sends them to the arithmetic unit at the proper time; and when the operation has been performed, still other control signals take the result from the arithmetic unit back to the memory. After an instruction has been executed, the control generator produces signals that enable the next instruction to go from the memory to the instruction decoder. In this way the instructions are performed *sequentially*.

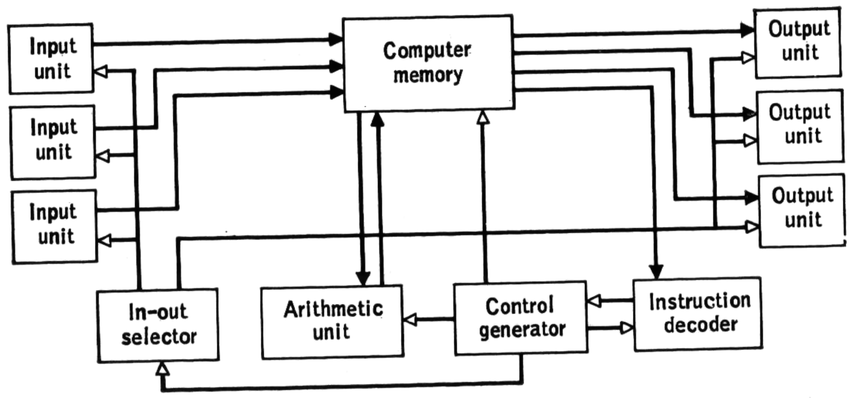


FIG. 1-20. Input-output.

Input-Output. There are many means by which numbers and instructions can be initially introduced into and finally read out of the computer memory. They can be typed directly into the computer on specially adapted electric typewriters, read into the computer by means of punched paper tape on a teletypelike machine, punched on cards and read into the computer by means of a punched-card machine, memorized on magnetic tape and read into the computer by a specially built tape recorder, etc. When the computations have been completed, the resulting numbers can be read out of the computer by having the computer operate a typewriter, punch paper tape, punch cards, put pulses on magnetic recording tape, etc. Most computers have several such *input units* and *output units*. The *in-out selector* determines which unit will read information into or out of the computer; it is controlled by signals from the control generator (see Fig. 1-20).

Summary. The computer's memory stores the instructions and the initial, intermediate, and final numbers involved in a computation. Instructions tell the computer what computations to perform. An instruction includes information about the kind of operation to be performed and the arguments involved in the operation.

The instruction decoder interprets an instruction. This interpretation

is sensed by the control generator. On the basis of that information, the control generator produces control signals that transmit the proper numbers from the memory to the arithmetic unit and that tell the arithmetic unit what operations to perform on these numbers or arguments. When the results have been computed by the arithmetic unit, the control signals guide them back to the memory. Then the control signals bring the next instruction into the instruction decoder, and the cycle continues. When new data is to be read into the memory by means of the input units, the control generator directs the in-out selector to choose the proper input unit. The control generator takes similar action when computed data is to be read out of the memory by means of one of the output units.

1-7. Functional Description of a Computer

The functional description of a computer involves consideration of further details about the computer's memory and the relation of an instruction to the memory. Two fundamental concepts of modern digital computers are involved in this relationship.

Words. The memory of a computer is composed of a large number of memory boxes, analogous to mailboxes. The information, or items to be memorized, is placed into these boxes as mail is placed into mailboxes. Each memory box, like a mailbox, has a name, or *address*. These addresses are usually numbers. For example, if the computer contains 100 memory boxes, their respective addresses might be the numbers from 1 to 100 (or 0 through 99). The information or items put into the memory boxes are called *words* and are analogous to the mail itself. However, *only one word can be stored in a memory box at a time.* As was seen in Sec. 1-6, there are two basic types of information, or words, that can be put into the boxes: words that are *numerical quantities* and words that are *computer instructions*.

At this point a short digression on computer terminology will prove helpful. The *memory box* of the previous paragraph is more correctly called an *address*, or alternatively a *memory address*, *memory location*, *storage location*, or *cell*. Instead of saying, "A word is in a memory box," computer personnel say, "The *contents* of an address is a word." Instructions are sometimes called commands; quantities are sometimes called data.†

An instruction word looks like a number, and there is no way to tell from the word itself whether it is a quantity or an instruction. The computer must be told explicitly which addresses contain instructions

† See First Glossary of Programming Terminology, *J. Assoc. Computing Machinery*, June, 1954; I.R.E. Standards on Electronic Computers: Definitions of Terms, 1956, *Proc. IRE*, vol. 44, no. 9, September, 1956.

and which contain quantities. Ordinarily a word is treated by the control as an instruction and by the arithmetic unit as a quantity. The fact that it is only the *interpretation of a word* that distinguishes instructions from quantities is the first fundamental concept of modern digital computers. This is fundamental because it enables the computer to *operate* on instructions. That is, the computer itself can formulate or change an instruction by treating it as though it were a quantity; and then, at a later time during the computations, this changed instruction can actually be executed. This gives the computer the capability of actually *writing* instructions as well as *executing* them; i.e., the computer can "tell" itself what to do—which is the closest that a computer probably ever comes to "thinking." More will be said about this in Part 2.

Instructions and Addresses. To see the relationship between an instruction and the computer's memory, consider, for example, an instruction that tells the computer to *add* two numbers. The instruction does *not* directly tell the computer to add the numbers themselves but rather tells the computer to add the numbers that are found as the *contents of addresses specified in the instruction.* If the numbers to be added are located in address α and address β, the instruction literally says, "Add the *contents of address* α to the *contents of address* β." Of course, the proper numbers to be added must have been previously placed into the addresses α and β. The same instruction might go on to tell the computer to put their sum into address γ and that the *next instruction* will be found as the contents of address δ. Hence, after our *add* instruction has been executed, the number that is the sum will be found as the contents of address γ. It must be certain that the contents of address δ is actually an instruction and not a number because the contents of address δ *will be interpreted* as if it were an instruction in any case.

The fact that instructions deal directly with addresses whose contents are to be operated on, rather than with the numbers themselves, is the second fundamental concept of digital-computing machines. This gives the computer an advantage that is exactly analogous to the advantage of performing algebra on symbols x, y, ...

As an example of an instruction, suppose that we wanted to add 125 and 412 and that we put 125 into address 32 and 412 into address 34. Suppose that we want the sum to appear in address 36 and that the next instruction is the contents of address 41. Then our *add* instruction, when decoded, would say, "Add the contents of address 32 to the contents of address 34, and put the sum into address 36; then the next instruction will be found as the contents of address 41."

The Accumulator. Consider now what happens in the arithmetic unit while an instruction is being executed. In most computers only one word at a time can be transferred between the arithmetic unit and the

memory. Hence, to perform an operation involving two arguments, the first argument must be transferred from the memory to the arithmetic unit and stored there temporarily while the second argument is being transferred. The special memory cell in the arithmetic unit for this purpose is called the *accumulator* (see Fig. 1-21). When the operation is performed, the result is formed in the accumulator before it is transmitted back to the memory.

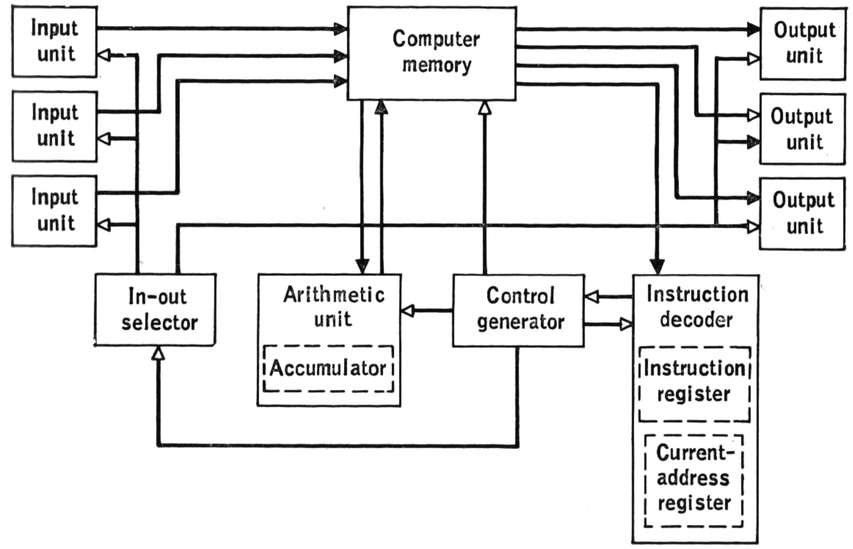


FIG. 1-21. Accumulator, instruction register, and current-address register.

Thus, in the above example, the contents of address 32, namely, 125, would be brought into the accumulator first; then this would be added to the contents of address 34, namely, 412, and the result, namely, 537, would be formed in the accumulator. Then 537 would be transmitted from the accumulator into address 36.

The Instruction Register and the Current-address Register. Next consider the instruction decoder that interprets an instruction. In order that the instruction decoder perform its function, it must constantly refer to the instruction being interpreted during the time control signals are being set up. To facilitate this, while an instruction is being executed it is stored in a special memory cell, called the *instruction register*, located in the instruction decoder (see Fig. 1-21).

There is another special memory cell located in the instruction decoder, called the *current-address register* (see Fig. 1-21). The contents of this register is (nearly) always the memory address from which the instruction being executed came. The reason for this is related to the fact that the address of the present instruction was given as part of the *previous* instruction.

For example, consider the *add* instruction described above. Suppose that this instruction were originally the contents of address 40. Then, while it is being executed, the contents of the instruction register will be the *add* instruction itself, and the contents of the current-address register will be 40. After the add operation has been executed (i.e., after the sum of 125 and 412 has been put into address 36) and just before the next instruction is read into the instruction register, the address of the next instruction, namely, 41, is recorded in the current-address register.

The Four Phases. The functioning of a computer during the execution of an instruction is often summarized in terms of four phases. Assuming that an instruction has already been transmitted into the instruction register, *phase* 1 involves the transmitting of the first argument from the memory into the accumulator. During this phase the instruction decoder determines the address of the first argument from the instruction, and the control generator produces control signals that transmit the contents of this memory address into the accumulator. During *phase* 2 the second argument is brought to the arithmetic unit and the operation is performed. Both these functions are included in the same phase because most often the operation is performed as the second argument enters the arithmetic unit. In this phase the instruction decoder determines from the instruction the address of the second argument and what operation is to be performed; the control generator generates signals that set up the arithmetic unit to perform this operation and signals that transmit the contents of the address of the second argument into the arithmetic unit; the arithmetic unit performs the operation, generally leaving the result as the contents of the accumulator. During *phase* 3 the contents of the accumulator is transmitted into the memory address specified by the instruction. During *phase* 4 the address of the next instruction is ascertained, and the contents of the current-address register is changed to the new address; the contents of this new address is then transmitted into the instruction register as the next instruction. Then phase 1 for this instruction is initiated, and the process is repeated.

It should be observed that computers are often designed with phase 1 or 3 omitted. That is, in some computers the memory address of both arguments need not necessarily be specified in a single instruction, nor need the memory address into which the result is transmitted always be specified. In such computers multiple instructions would be equivalent in function to a single instruction as described above. This aspect of instructions is discussed in considerable detail in Chap. 3, and we shall not press the point here, except to note that, although the principles described hold in general, the specific details vary with different computers.

Summary. There must be an instruction in the instruction register initially so that the whole process may begin, and there must be a way to stop the computer. These details will be considered in Chap. 3. The important points we wish to emphasize in this section are these: (1) *instructions and quantities* are memorized by the computer as the contents of memory *addresses;* (2) they are indistinguishable from each other except by interpretation, and (3) instructions explicitly involve only addresses and tell the computer what to do with the *contents* of these addresses; (4) the functioning of a computer during the execution of an instruction can be summarized in terms of the four phases as described.

1-8. Words and Pulses

Binary Numbers and Bits. Words are memorized in the computer as *binary numbers.* Hence it is necessary to divert our attention for a moment to a description of the binary notation for numbers. Numbers written in the binary system make use of only the symbols 0 and 1, called *bits,* just as numbers in the decimal system use the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, called *digits.* We say, for example, that 1101 is a binary number composed of 4 bits, just as 8,675 is a decimal number composed of 4 digits. In the decimal system the number 8,675 really means $(8 \times 10^3) + (6 \times 10^2) + (7 \times 10^1) + (5 \times 10^0)$, where $10^0 = 1$; analogously, in the binary system the number 1101 really means $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$, where $2^0 = 1$. Since $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, and $2^0 = 1$, 1101 is $(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 13$ in decimal. A more general and thorough discussion of the binary-number system appears in Chap. 2. Our purpose here is simply to show that binary numbers are made up of bits, 0 and 1, and that a binary number is just another notation for our familiar decimal numbers.

Words as Pulse Trains. As can probably be foreseen by the reader at this point, the reason for using the binary numbers in digital computers is to enable the use of simplified electronic circuitry. For then only two signal voltage levels need be distinguished, the voltage level that is to represent 1 and the voltage level that is to represent 0. A word is memorized by the computer as a series, or train, of 1 and 0 signal voltage levels, often called *pulses,* that represent the bits of the word. The number of bits in a single word is called the *length* of the word. All the words within a particular computer are of the *same length;*† hence, so are the contents of all addresses in the computer memory. When the proper signals are generated by the instruction decoder, the train of pulses that represents

† Modern computers are being developed to use different word lengths in different sections of the machines.

a word goes from the memory address to the arithmetic unit. In some computers this is accomplished in a serial manner in time; i.e., the pulses flow past a fixed point at some fixed rate, say 1 pulse/μsec. The *speed* of a computer is usually denoted by this pulse rate, or equivalently by the pulse frequency, i.e., 1 megacycle for our illustration.

It is often important to be able to denote a particular bit in a word by means of its position in the word. For this purpose we number the positions from *right to left.* Then Pn represents the nth bit of a word. For example, consider the 30-bit word in Fig. 1-22. Here $P1 = 0$, $P2 = 0$, . . . , $P6 = 1$, $P7 = 0$, $P8 = 1$, . . . , $P23 = 1$, $P24 = 0$, etc.

P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P 9	P 8	P 7	P 6	P 5	P 4	P 3	P 2	P 1
1	0	1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0	0	0

FIG. 1-22. Example of a 30-bit word.

Time Synchronization. In the arithmetic unit two word pulse trains are operated on by electronic circuits to form a new word pulse train that represents the result of the operation. In the control the word pulse train is decoded and interpreted as an instruction by other electronic circuits. The circuits that accomplish these functions are called *gates*, or digital switching circuits. Everything that happens to a word during the four phases takes place in these switching circuits, or gates.

It has been implied in the above discussion that there is some kind of synchronization of the pulses in a computer. In most computers this is indeed the case: all pulses are synchronized with respect to each other by means of *clock pulses* generated by a clock within the computer. The clock is an oscillator, with a frequency that is usually some integral multiple of the speed of the computer. Every gate is so connected to the clock of the computer that pulses are allowed to enter or leave the gates only at those times at which a clock pulse is received. The effect is analogous to an orchestra where the conductor beats out the timing, like the clock—the individual musicians, the gates, play notes, the pulses, according to this rhythm as directed by their music sheets, the design of the gating connections. At each beat of the clock all the pulses advance in the computer according to the gating connections, just as the musicians go on to the next musical beat. The purpose of this is to keep the phases of the pulses coordinated.

Example of Adding. Let us consider what happens in the arithmetic unit when two numbers are added. To be specific, suppose that these

numbers (where An and Bn label the bit positions) are

A30	A29	A28	A27	A26	A25	A24	A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A 9	A 8	A 7	A 6	A 5	A 4	A 3	A 2	A 1
0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1	0	0	1	1

B30	B29	B28	B27	B26	B25	B24	B23	B22	B21	B20	B19	B18	B17	B16	B15	B14	B13	B12	B11	B10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1
0	0	0	0	0	1	0	1	1	1	0	1	1	1	0	1	0	1	1	1	1	0	1	0	1	0	0	0	0	0

(The decimal equivalents are, for A, 46,202,483 and, for B, 24,600,224. See Chap. 2 for decimal-to-binary conversion methods.) Suppose that we are in phase 2; i.e., the first argument has already been transmitted into the accumulator. Suppose also that the pulse rate of the computer is 1 μsec; then the bits advanced once each microsecond. Figure 1-23 shows the situation at the start of phase 2; Fig. 1-24 shows the situation 1 μsec later; Fig. 1-25 after 2 μsec; Fig. 1-26 after 10 μsec; Fig. 1-27 after 20 μsec; and Fig. 1-28 after 30 μsec. As the two arguments advance

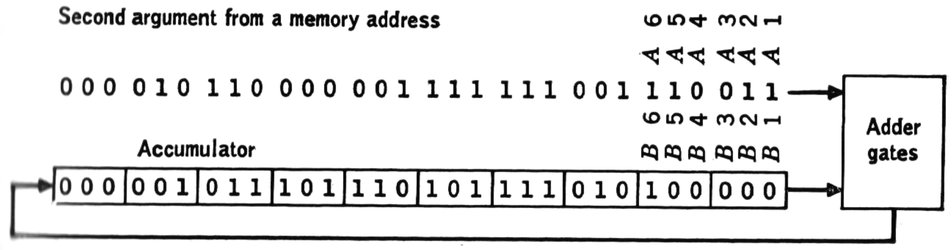


FIG. 1-23. Start of phase 2.

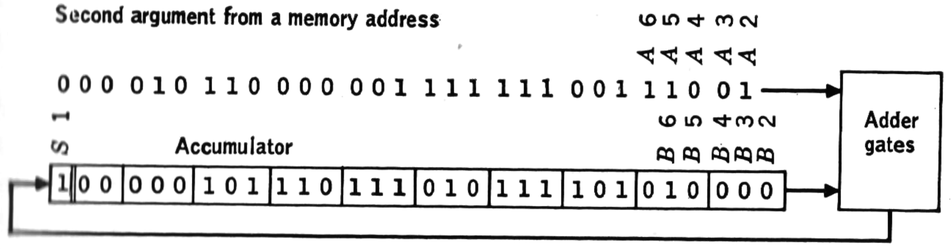


FIG. 1-24. One microsecond after start of phase 2.

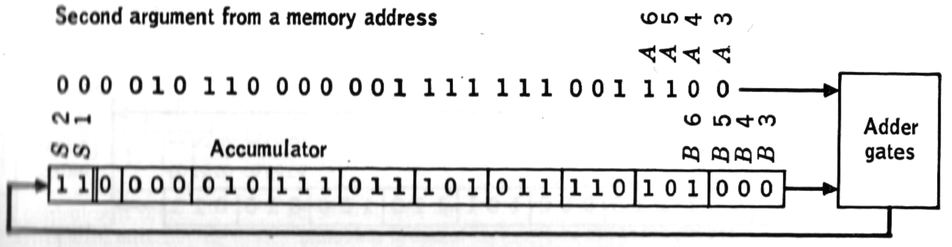


FIG. 1-25. Two microseconds after start of phase 2.

bit by bit into the adder, the result is formed bit by bit; the partial results are stored in the accumulator behind the partially used first argument as shown. The double line separates the partial first argument from the partial result so far formed in the accumulator. (The partial result is to the left of the double line; the remaining first argument is to its right.) In Fig. 1-28 the complete result has been formed in the accumulator, namely,

$S30$ $S29$ $S28$	$S27$ $S26$ $S25$	$S24$ $S23$ $S22$	$S21$ $S20$ $S19$	$S18$ $S17$ $S16$	$S15$ $S14$ $S13$	$S12$ $S11$ $S10$	$S\,9$ $S\,8$ $S\,7$	$S\,6$ $S\,5$ $S\,4$	$S\,3$ $S\,2$ $S\,1$
0 0 0	1 0 0	0 0 1	1 1 0	0 0 0	1 0 1	1 1 0	1 0 0	0 1 0	0 1 1

(which in decimal is 70,802,707). The computer is then ready to start on phase 3.

Figures 1-23 to 1-28 illustrate what would happen in a so-called "serial" computer. However, there are other, "parallel" methods for adding, in which all the digits of the sum are formed at the same time. Our purpose here was merely to give a general idea of how word pulse trains travel through gates in a synchronous fashion during a computation.

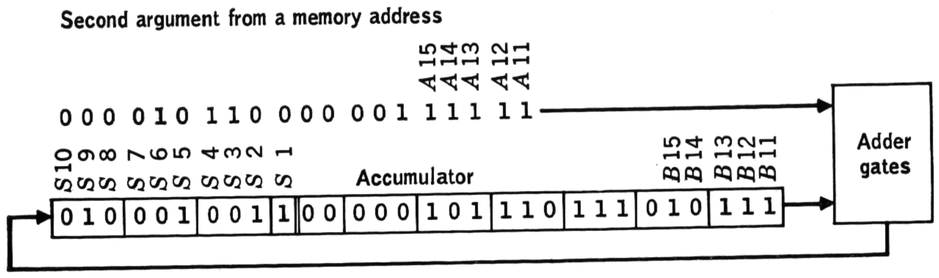


FIG. 1-26. Ten microseconds after start of phase 2.

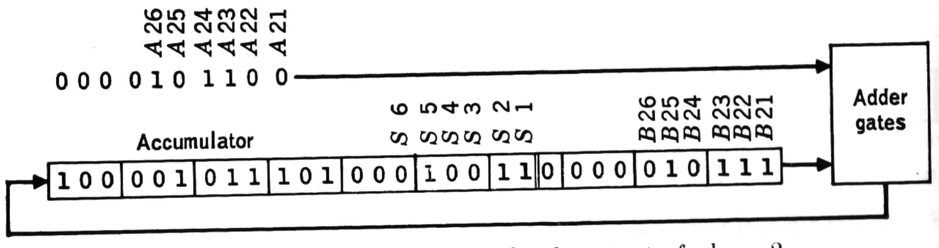


FIG. 1-27. Twenty microseconds after start of phase 2.

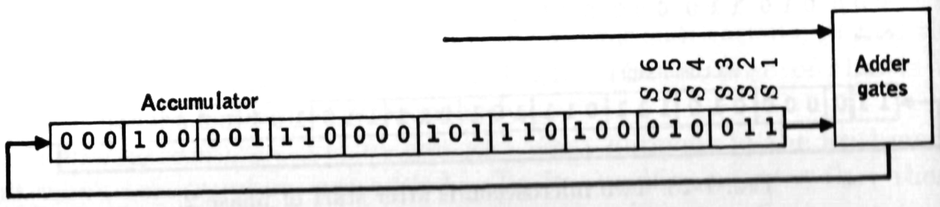


FIG. 1-28. Thirty microseconds after start of phase 2.

Instruction Decoding. We have given an example of how numerical quantities are gated. Next let us illustrate the method by which an instruction word is written and decoded. Suppose that the words of a hypothetical computer are each 30 bits in length, as above. Let the leftmost six bits, $P30$, $P29$, $P28$, $P27$, $P26$, $P25$, represent the operation code, i.e., tell what the instruction is to do: add two numbers, or form the product of two numbers, etc. Suppose, for concreteness, that the binary number 101 011 is the code for "add" (see Fig. 1-29). Let

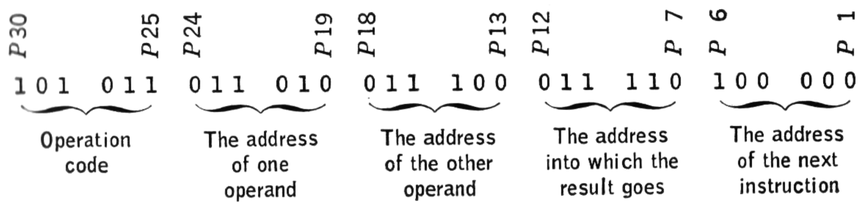


FIG. 1-29. Example of an instruction word.

the next six bits, $P24$, . . . , $P19$, represent the address of one of the numbers to be added; $P18$, . . . , $P13$, the address of the other number; and $P12$, . . . , $P7$, the address in which the sum is to be stored. The last six bits, $P6$, . . . , $P1$, tell where the next instruction is located in the memory. When the instruction word goes into the instruction decoder, it is stored in the instruction register. This register has a circuit for each bit being stored, with an output wire so that, if a 1 is stored, the circuit puts out the 1 signal voltage level on the wire, and the 0 signal voltage level if a zero. The outputs on these wires are sensed by gating circuits, and the instruction is thus decoded (see Fig. 1-30).

The control unit generates control pulses which are also *binary in character*, i.e., are composed only of the 0 and 1 voltage levels. In fact, almost all the circuitry in a digital computer is concerned with only these two voltage levels. This distinction can be made as to the function of pulses in the computer: some pulses represent words (numbers or instructions); other pulses represent control signals. Such a distinction was made in the block diagram of a computer, where solid-headed arrows represented the flow of word pulses and hollow arrows represented the flow of control-signal pulses.

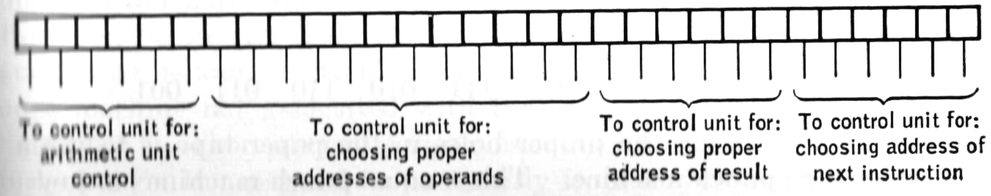


FIG. 1-30. Instruction register.

1-9. Input, Output, and Memory Systems

Input Information. As we have seen, there are many ways to get information into and obtain information from a digital computer. In order to give a more concrete picture of the process, we shall describe in some detail the procedures a person would follow in order to put information into a computer. The input information usually consists of a list of numbers and a list of instructions telling the computer what operations to perform on these numbers. For example, a problem might be to compute the trajectory of a missile (i.e., to construct a table giving the position of the missile at each second during its flight) given its initial velocity and angle of take-off. Suppose that the equation of the trajectory is

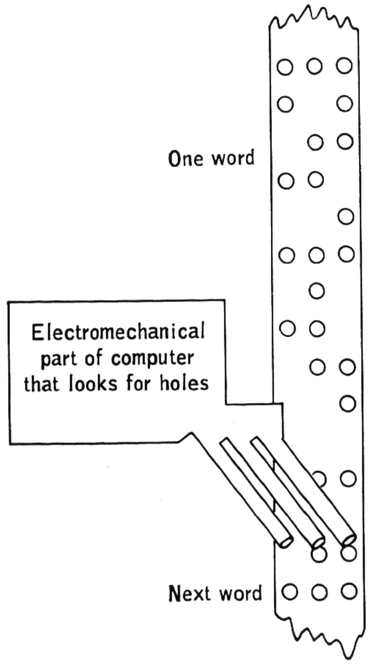


FIG. 1-31. Punched paper tape.

$$x = (V_0 \cos \theta)t \qquad y = (V_0 \sin \theta)t - \tfrac{1}{2}gt^2$$

where x,y = coordinates of missile at any time
V_0 = initial velocity
θ = angle of take-off from horizontal

Among the list of numbers that must be entered into the computer are the values of V_0, cos θ, sin θ, g, and the constant 1 (the time interval of 1 sec). A list of instructions must also be read into the computer telling it how to compute x and y. All these numbers and instructions must be entered into the computer in binary form. The first problem under consideration in this section is just how this might be accomplished.

Suppose that punched paper tape is to be used for this process. An electromechanical device exists so that a hole in the tape is interpreted by the computer as a 1, no hole as a 0. Consider a paper tape with positions for three holes per line. If there are 30 bits to a word, then 10 lines are needed per word. Figure 1-31 illustrates how the following word would look on a tape:

111 101 011 110 001 111 010 110 011 001

One method for getting the proper holes in the paper tape is to use a hand paper-tape punch machine. This manual punch machine has keys: pushing a single key will punch one line. The keys might be numbered

from 0 to 7, and the array of holes corresponding to each key might be as given in Fig. 1-32. Hence the word above would be punched by pushing the following keys in succession:

7, 5, 3, 6, 1, 7, 2, 6, 3, 1

The person who prepares the list of numbers and instructions for a problem is called a *programmer*, or *coder*, and the list itself is called a *coding sheet*. The *code* is written as a list of computer words. However, the coder does not write the words in binary form but instead puts down, as a shorthand, the numbers of the keys that are to be punched. Hence instead of the word above, the coding sheet would have

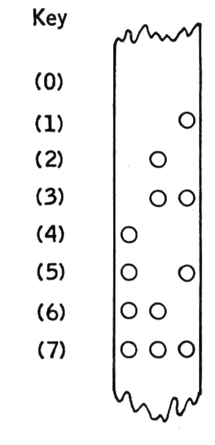
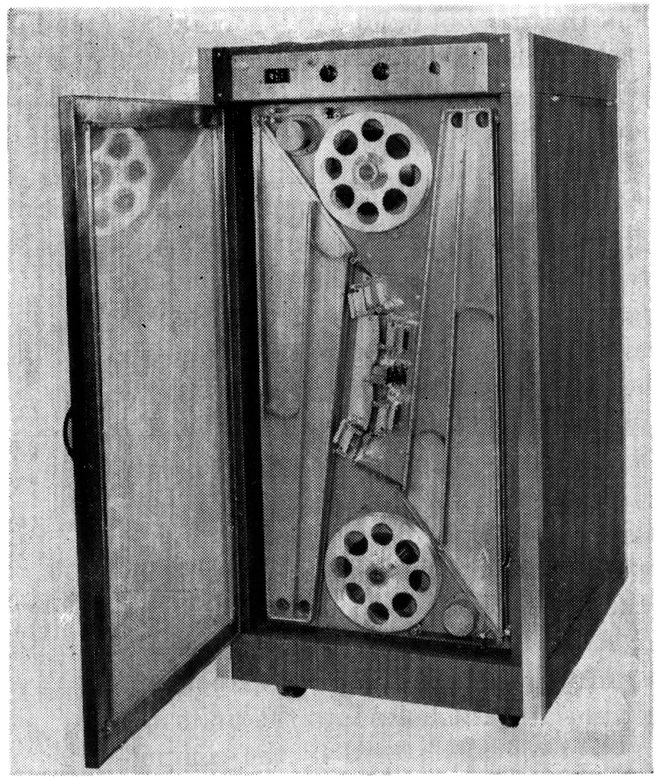


FIG. 1-32. Punched-paper-tape code.

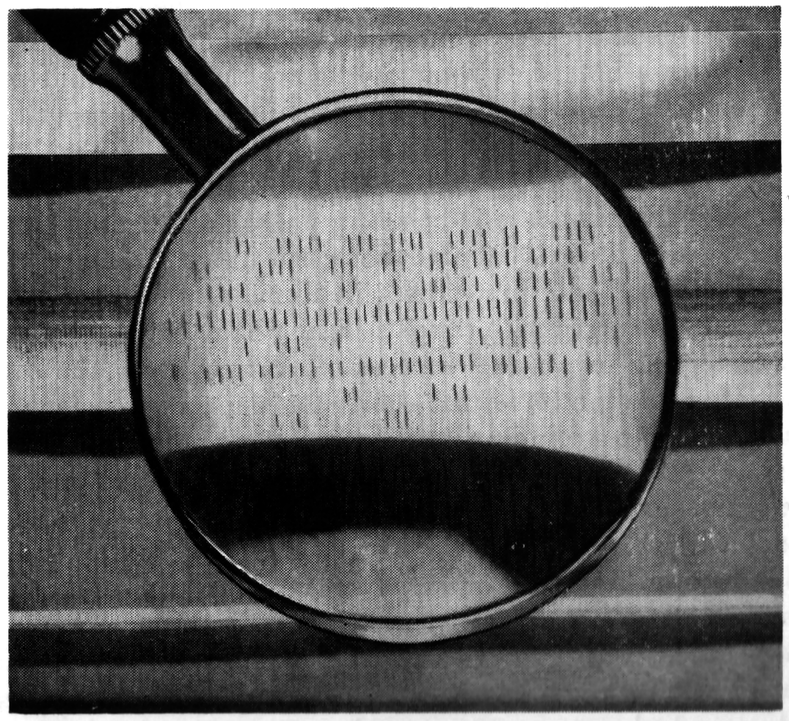
75 36 17 26 31

The punch operator would push these keys in succession, and the paper tape would be punched in a binary mode. After the punching is complete, the paper tape is run through the computer paper-tape reader, which translates the holes into the units and zeros that are electronically stored in the computer's memory. The computer then calculates the results, and they are read out—perhaps by having the computer punch a paper tape. This paper tape would then be put into a machine that does the reverse of punching, i.e., that looks at a paper tape and prints on paper the numbers that appear there, in a manner similar to a ticker-tape machine.

Other Input-Output Methods. Besides punched paper tape there are punched cards: the cards are punched in much the same manner as the paper tape, and the device that reads the cards into the computer likewise senses the holes and interprets these in terms of electronic pulses. A typewriter may be rigged with relays so that when a key is pushed a code is produced by the relays and transmitted directly into the computer. Such methods are often found too slow. A faster method for putting a program or code into a computer is by means of magnetic tape, like sound-recording tape (see Fig. 1-33); here a small magnetized area on the tape is interpreted by the computer as a unit, no magnetized area as a zero. However, before this magnetic tape is read into the computer, the program must be first "written" onto the tape. This may be accomplished on a machine not associated with the computer itself, often called an "inscriber." There are many varieties of inscribers; some enable the code to be put on the magnetic tape directly from a typewriter, others from previously punched paper tape or cards. For example, one procedure



(*a*)



(*b*)

FIG. 1-33. (*a*) Magnetic-tape unit; (*b*) magnetic tape, showing locations of bits. (*Photographs courtesy Potter Instrument Company, Inc.*)

may require that the code first be punched on paper tape and the paper tape "inscribed" onto magnetic tape, which is finally read into the computer. Table 1-1 gives some idea of the speeds in bits per second at which information can be transformed from one medium to another. Knowing the number of bits per word, one can then calculate the number of words per minute that can be so transformed.

TABLE 1-1. ORDER OF MAGNITUDE OF RATES

From	To	Speed
Paper tape	Computer memory	2,000 bits/sec
Punched cards	Computer memory	3,000
Typewriter	Computer memory	40
Magnetic tape	Computer memory	100,000
Paper tape	Magnetic tape	2,000
Punched cards	Magnetic tape	3,000
Typewriter	Magnetic tape	40
Computer memory	Paper tape	300
Computer memory	Punched cards	2,000
Computer memory	Magnetic tape	100,000
Computer memory	Typewriter	600 characters/min
Paper tape	Typewriter	600
Punched cards	Printer	10,000
Magnetic tape	High-speed printer	100,000
Computer memory	High-speed printer	100,000
Computer memory	Cathode-ray-tube printer	500,000

There are also some unusual means for putting information into the computer. One is a reading machine that can translate digits and letters, as they appear on a printed page, into a binary code and write this code into the memory.

Most of the methods for taking information from the outside world into the computer's memory can be reversed to serve as methods for reading information out from the computer's memory to the outside world in a form that is readily understandable. Hence the computer can work a typewriter, punch paper tape, punch cards, or write onto magnetic tape. Of course the "*out*scriber" is required to transform the information from punched paper tape, punched cards, or magnetic tape into the letter and number characters we are more familiar with. In addition there are methods using so-called "high-speed" printers and specially made cathode-ray tubes (see Figs. 1-34 to 1-36) that like the typewriter display numbers and letters directly from the computer.

Other outputs from a computer may be obtained from attaching oscilloscope, or television, tubes to the computer's memory. Then the

computer literally can draw maps, can indicate the movements of dots representing aircraft, etc., or can display other kinds of pictures.

Memory Systems. Words read into the computer go into the computer's memory. The function of a computer memory is to store the pulses of each word in such a way that they may be called into the arithmetic unit rapidly, upon generation of the proper control signals. In

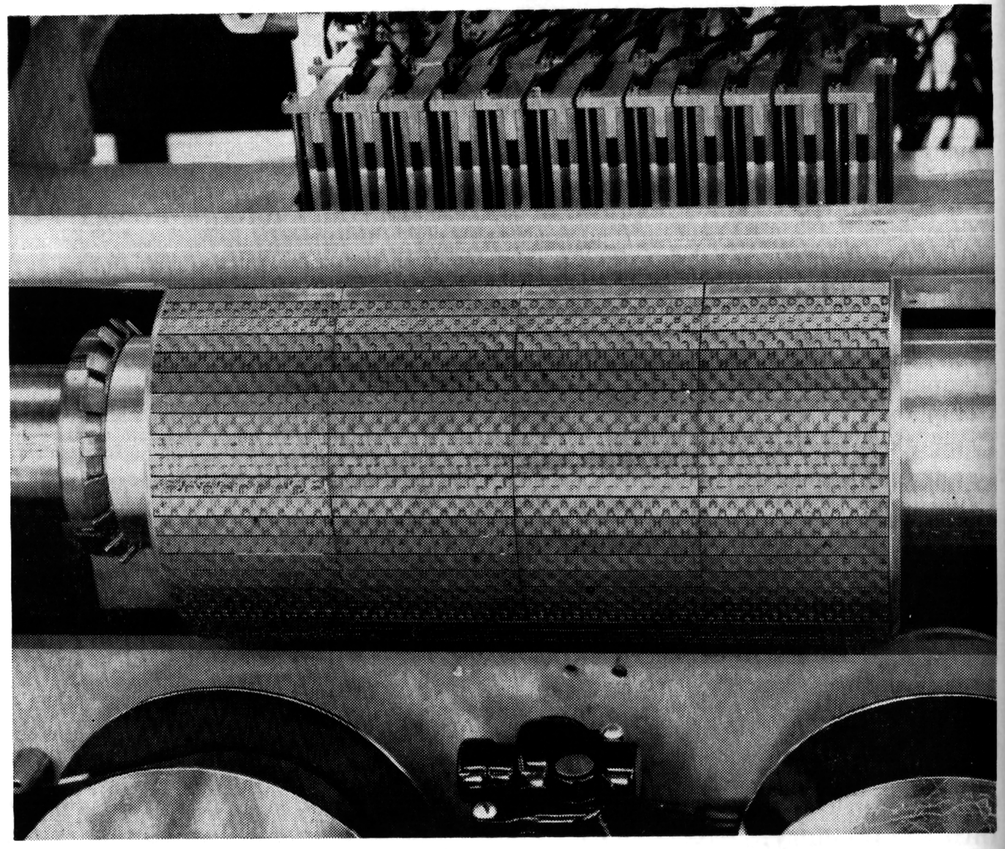


FIG. 1-34. High-speed printer showing type-face drum. As the drum rotates, a hammer pushes the paper up against the proper letter in each column. In this way an entire line is printed at one time during each revolution of the drum. (*Photograph courtesy Potter Instrument Company, Inc.*)

addition, the computer memory must be able to accept rapidly the pulse train of a word for storage. The speed of modern computers is essentially limited by the speeds of access to their memories.

An important point that is true for practically all kinds of memories is that when a word is *read out of a memory* the word is *still stored in the memory* (unless otherwise specifically stated by the instruction—see Chap. 3). On the other hand, when a word is written *into the memory* from the arithmetic unit, *the previous word of that address is automatically erased and replaced by the new word.*

There are many ways of memorizing words in a digital computer. Magnetic tape, described above as an input-output system, can as well be used for a computer memory. A similar magnetic-pulse technique is incorporated into magnetic drums, which are simply rapidly rotating



FIG. 1-35. Sample print-out from a high-speed printer. This printer uses a type-face chain, as shown, instead of a drum. The chain moves horizontally behind the paper, which moves vertically. When the proper character is aligned with a given position, a hammer strikes an inked ribbon against the paper and type face. (*Photograph courtesy International Business Machines Corp.*)

cylinders with magnetizable surfaces (see Fig. 1-37). Faster memories are constructed from magnetic cores, specially prepared from ferrites having a square hysteresis loop, thereby presenting two stable states. Each core then represents one bit of a word (see Fig. 1-38). An even

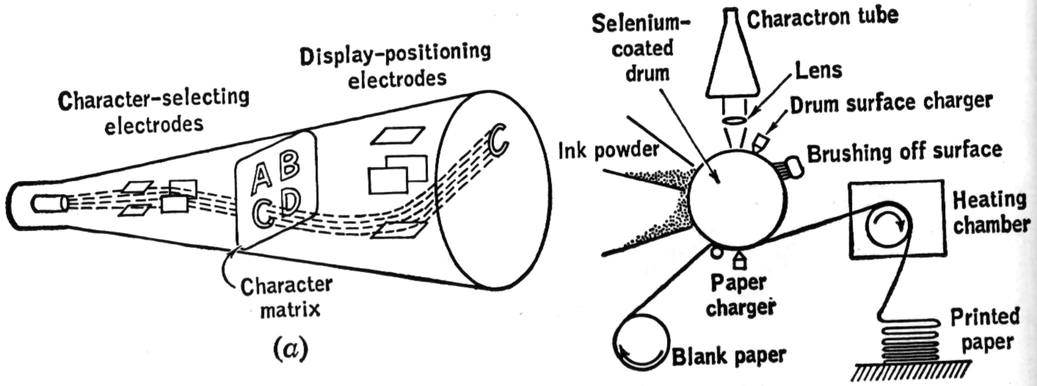


FIG. 1-36. The Charactron tube coupled with the xerography process for high-speed printing. (*a*) The cathode-ray tube, showing the character-selecting electrodes which send the electron beam through the proper hole in the character-forming matrix; (*b*) the light image from the cathode-ray tube sensitizes the selenium-coated drum, which then rotates to transfer the ink powder, which sticks to the sensitized area of the paper.

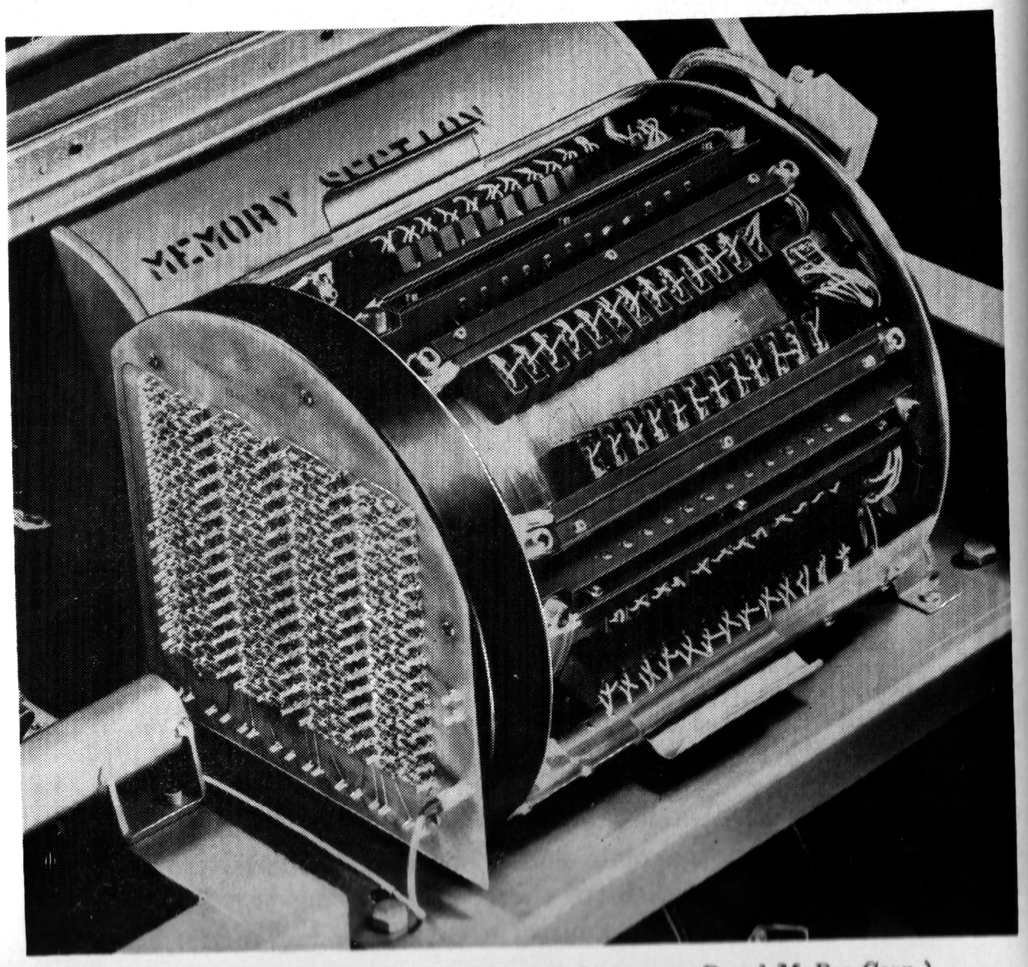


FIG. 1-37. Magnetic drum. (*Photograph courtesy Royal McBee Corp.*)

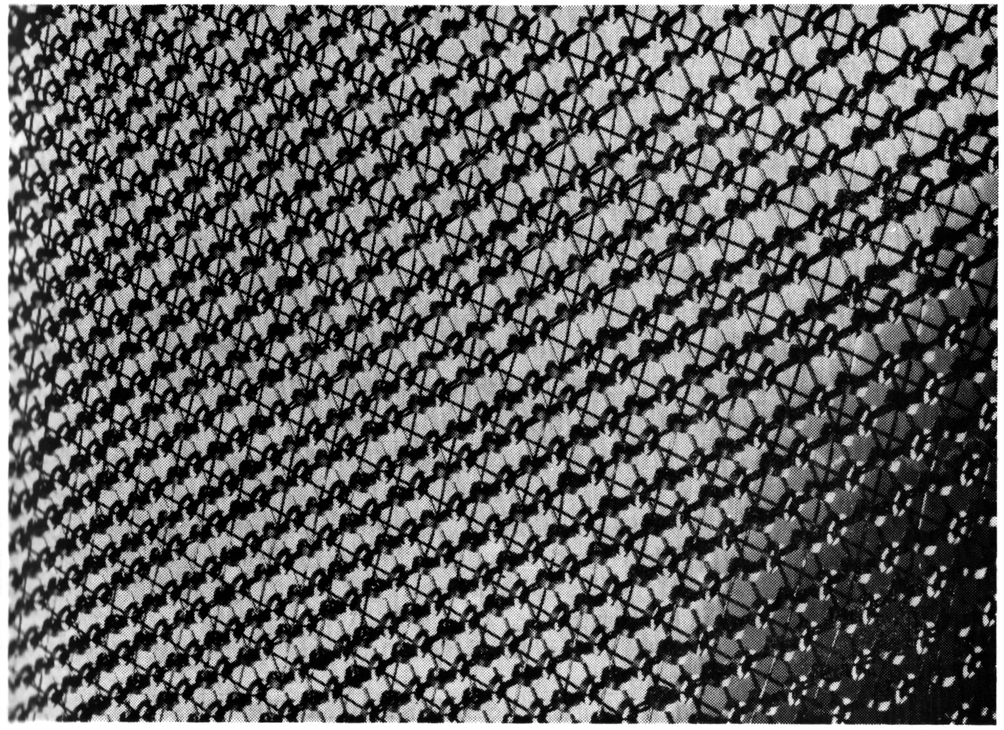


FIG. 1-38. Magnetic cores. (*Photograph courtesy International Business Machines Corp.*)

faster memory has been constructed, using diodes and capacitors. The memory with the fastest reaction time is one constructed using a type of gating circuit, called a "flip-flop," to record each bit. The time required to transmit one computer word out of the memory to where it will be used is called the *memory access time* (see Table 1-2).

TABLE 1-2. ORDER OF MAGNITUDE OF MEMORY ACCESS TIME

Memory system	*Time required to transmit a 30-bit word*
Magnetic tape	5 msec + time necessary to position tape
Magnetic drum	10 msec
Acoustic delay line	100 μsec
Magnetic core	5 μsec
Diode capacitor	1 μsec
Flip-flop register	1 μsec

In the magnetic-tape memory the whole tape must be searched, and a great deal of time is lost just winding and rewinding tape. For the drum, the computer must wait for the word to rotate to the proper read-out position. The magnetic-core arrangement enables any word to be read out with no delays except those inherent in the circuitry involved. A word is read from a core memory in parallel; i.e., all the bits are transmitted to the arithmetic unit at the same time. A similar situation

occurs with diode-capacitor memory or with a memory composed of a set of flip-flop registers.

Unfortunately, the faster the memory, the more it costs per bit. Hence a computer usually has several kinds of memory. Most commonly, magnetic tapes and drums are used for the so-called "low-speed" memory of a computer, while a magnetic-core memory or diode-capacitor memory is used as the "high-speed" memory. Often there is also included in the larger computers a battery of "flip-flop" registers for the really high-speed work. In this way, various parts of a large program can be stored in the relatively low-speed memory system of a computer and then transmitted to the high-speed memory when the actual computations of this part of the program are to occur.

1-10. Exercises

1-1. What are the general purpose and function of an operation? An instruction? The computer memory? The computing unit? The arithmetic unit? The control? The instruction decoder? The control generator? Control signals? Input units? Output units? The in-out selector?

1-2. If 2 and 3 are added to obtain 5, what are the arguments of this operation? The result?

1-3. Write a sequence of instructions by means of which $a(b + cd)/f$ can be computed. Letting $a = 6$, $b = 4$, $c = 2$, $d = 3$, and $f = 5$, perform the sequence of instructions.

1-4. Write a sequence of instructions by means of which $(1 - a^2)^3$ can be computed.

1-5. What *four* functions do the control signals perform?

1-6. Discuss the first two fundamental concepts of modern digital computers, using accepted computer terminology.

1-7. Consider the example of Sec. 1-6. Suppose that the value of a is found as the contents of address 50, of b as the contents of address 51, and of c as the contents of address 52; the value of the intermediate result x is to be put into address 53, and the final result y is to be put into address 54. How would instructions 1 and 2 of that example be written if these instructions themselves were found as the contents of addresses 55 and 56, respectively?

1-8. Describe by naming specific addresses and special memory cells what happens during each of the four phases during the execution of the instruction given as an example on page 32. What is the contents of the instruction register? What is the contents of the current-address register? What is the contents *of the contents* of the current-address register? (HINT: The contents of the current-address register is an address itself.)

1-9. During phase 1 what is the relation between the contents of the instruction register and the contents of the current-address register?

1-10. Define: length of word; pulses; speed of computer; gates; clock pulses.

1-11. In the addition example given in Sec. 1-8, what is the contents of the accumulator 15 μsec after the start of phase 2? 25 μsec after?

1-12. What keys of a paper-tape puncher would be pushed to punch the number

101 110 011 001 101 101 011 011 001 100

(Use the table of Fig. 1-32.)

1-13. Draw a picture of paper tape with the number in Exercise 1-12 punched on it.

1-14. What number does the hole pattern of Fig. 1-39 represent (left to right)?

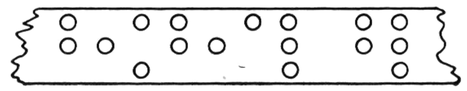


FIG. 1-39

REFERENCES

The following are general books about computers that contain examples of their use:

Alt, Franz L.: "Electronic Digital Computers—Their Use in Science and Engineering," Academic Press, Inc., New York, 1958.

Bellman, R.: "Adaptive Control Processes," Princeton University Press, Princeton, N.J., 1961.

Booth, A. D., and K. H. V. Booth: "Automatic Digital Computers," Academic Press, Inc., New York, 1956.

Bowden, B. V.: "Faster than Thought," Pitman Publishing Corporation, New York, 1953.

Canning, R. G.: "Electronic Data Processing for Business and Industry," John Wiley & Sons, Inc., New York, 1956.

Chapin, Ned: "Automatic Computers—A System Approach for Business," D. Van Nostrand Company, Inc., Princeton, N.J., 1959.

Eckert, W. J., and Rebecca Jones: "Faster, Faster," McGraw-Hill Book Company, Inc., New York, 1956.

Edmundson, H. P., et al.: Studies in Machine Translation, *Repts.* 1–9, Rand Corp., 1958.

Gass, Saul I.: "Linear Programming," McGraw-Hill Book Company, Inc., New York, 1958.

Gotlieb, C. C., and J. N. P. Hume: "High-speed Data Processing," McGraw-Hill Book Company, Inc., New York, 1958.

Gregory, R. H., and R. L. Van Horn: "Automatic Data-processing Systems," Wadsworth Publishing Co., San Francisco, 1960.

Householder, A. S., G. E. Forsythe, and H. H. Germond: Monte Carlo Method, *Natl. Bur. Standards (U.S.) Appl. Math. Ser.*, no. 12, 1951.

Howard, R. A.: "Dynamic Programming and Markov Processes," Technology Press and John Wiley & Sons, Inc., New York, 1960.

Ledley, R. S.: "Use of Computers in Biology and Medicine," McGraw-Hill Book Company, Inc., New York (in press).

Levin, H. S.: "Office Work and Automation," John Wiley & Sons, Inc., New York, 1957.

Livesley, R. K.: "Digital Computers," Cambridge University Press, New York, 1957.

Locke, W. N., and A. D. Booth (eds.): "Machine Translation of Languages," John Wiley & Sons, Inc., New York, 1955.

McCormick, E. M.: "Digital Computer Primer," McGraw-Hill Book Company, Inc., New York, 1959.

McCracken, D. D., H. Weiss, and Tsai-Hwa Lee: "Programming Business Computers," John Wiley & Sons, Inc., New York, 1959.

Mandl, Matthew: "Fundamentals of Digital Computers," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1958.

Wilkes, M. V., D. J. Wheeler, and S. Gilli: "The Preparation of Programs for an Electronic Digital Computer," 2d ed., Addison-Wesley Publishing Company, Reading, Mass., 1957.

Wrubel, M. H.: "A Primer of Programming for Digital Computers," McGraw-Hill Book Company, Inc., New York, 1959.

The following are periodicals that often contain examples of the use of computers:

Communications of the ACM, monthly, Association for Computing Machinery, New York.

Computer Journal, British Computer Society, Ltd., London.

Computers and Automation, Edmund C. Berkeley and Associates, New York.

Computing News, J. W. Granholm, Seattle.

Computing Reviews, Association for Computing Machinery, New York.

Control Engineering, monthly, McGraw-Hill Publishing Company, Inc., New York.

Data Processing Digest, Canning, Sisson, & Associates, Los Angeles.

Datamation (*Research and Engineering, The Magazine of Datamation*), bimonthly, Relyea Publishing Co., New York.

IBM Journal of Research and Development, International Business Machines Corporation, New York.

IRE Transactions on Electronic Computers, Professional Group on Electronic Computers, Institute of Radio Engineers, Inc., New York.

Journal of the Association for Computing Machinery, New York.

Journal of Operations Research Society of America, Baltimore.

Journal of the Society for Industrial and Applied Mathematics, Philadelphia.

Machine Translation, Massachusetts Institute of Technology, Cambridge, Mass.

Mathematical Tables and Other Aids to Computation, National Academy of Sciences–National Research Council, Washington, D.C.

Proceedings of the . . . Joint Computer Conferences of the Professional Group on Electronic Computers (IRE), the Association for Computing Machinery, and the American Institute of Electrical Engineers (AIEE) Committee on Computing Devices.

The engineering details of digital electronic computers can be found in:

Ledley, R. S.: "Digital Computer and Control Engineering," McGraw-Hill Book Company, Inc., New York, 1960.

The following are selected articles dealing with the application of computers involving the solution to equations:

Barlow, J. S.: Autocorrelation and Crosscorrelation Analysis in Electroencephalography, *IRE Trans. on Med. Electronics*, vol. ME-6, no. 3, pp. 179–183, 1959.

Berman, M., and R. L. Schoenfeld: Information Content of Tracer Data with Respect to Steady State Systems, in *Symposium on Inform. Theory in Biology*, 1958.

Cole, K. S., H. A. Antosiewicz, and P. Rabinowitz: Automatic Computation of Nerve Excitation, *J. Soc. Ind. Appl. Math.*, vol. 3, no. 3, p. 153, 1955; Correction, *ibid.*, vol. 6, no. 2, p. 196, 1958.

Farley, B. G., and W. A. Clark: Simulation of Self-organizing Systems by Digital Computer, *Trans. IRE* (*Inform. Theory*), vol. PGIT-4, pp. 76–84, September, 1954.

Hovland, Carl I., and Earl B. Hunt: Computers in Behavioral Science, *Behavioral Sci.*, vol. 5, pp. 265–273, July, 1960.

Issacson, E., J. J. Stoker, and A. Troesch: Numerical Solution of Flow Problems in Rivers, *J. Hydraulics Division*, vol. 84, pp. 1810–1818, October, 1958.

Kirsch, R. A., L. Cahn, C. Ray, and G. H. Urban: Experiments in Processing Pictorial Information with a Digital Computer, *Proc. Eastern Joint Computer Conf.*, December, 1957, Washington, D.C., pp. 221–229.

Ledley, R. S.: Digital Electronic Computers in Biomedical Science, *Science*, vol. 130, no. 3384, pp. 1225–1234, Nov. 6, 1959.

——— and L. B. Lusted: Reasoning Foundations of Medical Diagnosis, *Science*, vol. 130, no. 3366, pp. 9–21, July 3, 1959.

——— and ———: The Use of Electronic Computers to Aid in Medical Diagnosis, *Proc. IRE*, vol. 47, pp. 1970–1977, November, 1959.

——— and ———: Computers in Medical Processing, *Operating Research*, vol. 8, no. 3, May–June, 1960.

——— and ———: The Use of Electronic Computers in Medical Data Processing: Aids in Diagnosis, Current Information Retrieval and Medical Record Keeping, *IRE Trans. on Med. Electronics*, vol. ME-7, pp. 31–47, June, 1960.

Livesley, R. K.: The Analysis of Large Structural Systems, *Computer J.*, vol. 3, pp. 34–38, April, 1960.

Potts, T. F., G. N. Ornstein, and A. B. Clymer: The Automatic Determination of Human and Other System Parameters, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 645–660.

Ruskin, V. W., J. H. Drinnan, and J. B. Claydon: Automatic Load Projection and Substation Planning by Computer, *Proc. Comp. Data Processing Soc. Canada*, June, 1960, pp. 193–210.

Simpson, H. R.: The Estimation of Linkage on an Electronic Computer, *Ann. Human Genetics*, vol. 22, pp. 356–361, 1958.

Taback, L.: Analog-Digital Conversion Equipment for Electrocardiographic Data, *Natl. Bur. Standards* (*U.S.*) *Tech. Note* 42, April, 1960.

———, E. Marden, H. L. Mason, and H. V. Pipberger: Digital Recording of Electrocardiographic Data for Analysis by a Digital Computer, *IRE Trans. on Med. Electronics*, vol. ME-6, pp. 167–171, September, 1959.

Weinrauch, H., and A. W. Hetherington: Computers in Medicine and Surgery, *J. Am. Med. Assoc.*, vol. 169, no. 3, pp. 240–245, Jan. 17, 1959.

——— and E. Pattishall: Air Force and Application of Computers to Medicine and Biology, *Military Med.*, vol. 122, no. 178, 1958.

Wrigley, C. F.: Electronic Computers and Psychological Research, *Am. Psychologist*, vol. 12, no. 501, 1957.

The following are selected articles dealing with the application of computers involving the control of processes:

Anon.: Monsanto Unveils Integrated Computer-controlled Process, *Instr. Control Systems*, vol. 33, no. 11, pp. 1888–1893, November, 1960.

Assuer, A.: Computers and Control Systems, *J. IEE*, vol. 6, no. 66, pp. 335–338, June, 1960.

Cannon, E. W.: Computer Keeps Tab on Working Process, *Control Eng.*, vol. 7, pp. 138–139, September, 1960.

Karp, Harry R.: Using Corporate Models for Business Control, *Control Eng.*, vol. 7, pp. 135–137, September, 1960.

Marton, Fred D.: Process Control Systems, *Instr. Control Systems*, vol. 33, no. 12, pp. 2064–2065, December, 1960.

Parsons, J. R., E. D. Tomin, and A. J. Andrews: Controlling a Thermal Cracking Furnace with Digital Computer, *Control Eng.*, vol. 7, pp. 150–153, September, 1960.

Reid, Marvin: Job Shop Production Control on the Computer, *Control Eng.*, vol. 8, pp. 107–110, January, 1961.

Teale, K.: The Development of Numerically Controlled Machine Tools, *Electronic Forum for Ind. Bull.*, vol. 1, no. 3, pp. 9-13, May, 1960.

Terekhov, G. A.: Main Trends in the Automation and Mechanization of the Machining Processes in Machine Building, *LLU Transl. Bull.*, vol. 2, no. 8, pp. 635–669, August, 1960.

The following are selected articles dealing with applications of computers involving simulations:

Clymer, A. B.: Direct System Synthesis by Means of Computers, *Trans. AIEE, Communs. and Electronics*, January, 1959, pp. 798–806.

Dempsey, J. M.: Some Aspects of Simulator Design, *Computer J.*, vol. 3, no. 3, pp. 158–161, October, 1960.

D'Esopo, D. A., H. L. Dixon, and B. Lefkowitz: A Model for Simulating an Air-transportation System, *Naval Research Logistics Quart.*, no. 3, pp. 213–220, September, 1960.

Dickie, H. F., and E. C. Throndsen: Manufacturing Systems Simulation, *Factory*, no. 118, pp. 114–117, October, 1960.

Harman, H. H.: Simulation: A Survey, *Proc. Western Joint Computer Conf.*, May, 9–11, 1961, Los Angeles, Calif., pp. 1–10.

Malcolm, D. G.: Bibliography on the Use of Simulation in Management Analysis, *Operations Research*, no. 8, pp. 169–177, 1960.

Moore, C. J., and T. S. Lewis: Digital Simulation of Discrete Flow Systems, *Communs. ACM*, vol. 3, pp. 659–660, December, 1960.

Shubik, M.: Bibliography on Simulation, Gaming, Artificial Intelligence and Allied Topics, *J. Am. Stat. Assoc.*, vol. 55, no. 292, pp. 736–751, December, 1960.

Soroka, Walter W.: Simulation in Science and Technology, *Appl. Mech. Rev.*, vol. 13, no. 9, pp. 621–624, September, 1960.

Youle, P. V.: Simulation of Full-scale Multi-state Batchwise Chemical Plant, *Computer J.*, vol. 3, no. 3, pp. 150–157, October, 1960.

The following are selected articles dealing with applications of computers involving data processing (see also References at end of Chap. 7):

Anon.: The Scope of Data-processing, *Data Processing*, vol. 2, pp. 31–38, April–May, 1960.

Anon.: The ADP Experience of Local and State Governments, *Automatic Data Processing*, vol. 2, no. 11, pp. 25–27, November, 1960.

Bean, Keith: Machines of Tomorrow, *Automatic Data Processing*, vol. 2, no. 9, pp. 24–25, September, 1960.

———: Machines like Minds? *Automatic Data Processing*, vol. 2, no. 10, pp. 20–22, October, 1960.

Fleishman, A. M.: Future Is Bright for ADP in Smaller Companies, *Automation*, vol. 8, no. 1, pp. 66–72, January, 1961.

Perry, M. N., and W. R. Plugge: American Airlines "SABRE" Electronic Reservation System, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 593–602.

Swanson, P. R.: Information Retrieval, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 239–248.

Teedle, John B.: Reducing Human Error in ADP Systems, *Datamation*, vol. 6, no. 6, pp. 91–94, November–December, 1960.

Warshaw, Martin J.: Optical Character Reading, *Retail Control*, vol. 29, no. 4, pp. 19–32, December, 1960.

CHAPTER 2

PRINCIPLES OF PROGRAMMING

2-1. Introduction

The purpose of the current chapter is to introduce the student to the fundamental *principles* of coding and programming, as opposed to the *tactics* of coding and programming. These principles are concerned with sequences of instructions, the coding of decisions, the flow-charting and symbolic-coding aids, and the use of loops and instruction modification. In order to leave the student free to concentrate on these principles, the four-address format, with a minimum number of instruction types, is utilized. This serves to separate consideration of the *principles* from that of the tactics. However, it should be pointed out that the four-address format is used in this chapter for pedagogical reasons only. Discussions of instruction formats in actual use on currently available computers, and of the large variety of instruction types utilized in actual practice, are associated with the *tactics* of coding and programming; these will be considered in detail in Chap. 3.

The terms *coding* and *programming* are often used as synonyms. However, a code is more specifically a short list of instructions that direct the computer to perform only a part of the entire calculation, whereas the term program refers to the complete list of instructions used for the problem. Hence the term programming usually includes the over-all planning of the use of the computer for a particular problem as well as the writing of the instruction lists, or codes, whereas coding is usually limited in meaning to just the writing of the instructions lists. Sometimes a code is called a *routine*.

Stages in Programming. As will be seen, there are five stages in programming. First the computations to be performed must be clearly and precisely defined. The over-all plan of the computations is diagramed by means of a so-called "flow chart." The second stage is the actual coding. It is often best to write a code in terms of a symbolic language first, for then changes are easily made. Numbers are assigned to the symbols, and the final code is prepared. In the third stage some procedure is used to get the code into the memory of the computer. The fourth stage consists in "debugging" the code, i.e., detecting and correcting any

errors. The fifth and final stage involves running the code on the computer and tabulating the results. In this chapter and the next, only the first two stages are considered. Techniques for the fourth stage are described in Chap. 5.

Philosophy of Programming. As will become abundantly clear in the following sections, a single error in one instruction invalidates the entire code. Hence programming is an exacting technique, requiring attention to details without losing sight of the over-all plan. Learning to program must be developed from within a person. Following through each instruction of the examples given in this chapter is important, but it is not sufficient. The student must program problems by himself if he is to become aware of all the detailed thinking and reasoning associated with the programming technique. The method of teaching in this text is through example. The student should try to do the examples himself, using the text only as a guide to the methods involved.

2-2. Number Systems: Conversion

Number Systems and the Radix. Preliminary to any discussion of coding should come an analysis of various number systems. In Sec. 1-8 we noted that 8,675 really means

$$8 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$$

where $10^0 = 1$. Since decimal numbers are expanded as illustrated in powers of 10, we say that the decimal number system has a *radix* of 10. This concept can be generalized to define number systems based on any positive-integer radix q: if a number N is based on radix q, then it can be expanded as

$$N = a_n q^n + a_{n-1} q^{n-1} + \cdots + a_2 q^2 + a_1 q^1 + a_0 q^0$$

for integer $N \geq 1$, or

$$N = a_{-1} q^{-1} + a_{-2} q^{-2} + a_{-3} q^{-3} + \cdots + a_{-m} q^{-m}$$

for $0 < N < 1$, where $a_0, a_1, \ldots, a_{-1}, a_{-2}, \ldots$ are nonnegative integers each less than q. Of course, for noninteger $N > 1$, the expansion contains both the positive and negative powers,

$$N = a_n q^n + \cdots + a_2 q^2 + a_1 q^1 + a_0 q^0 + a_{-1} q^{-1} + a_{-2} q^{-2} + \cdots + a_{-m} q^{-m}$$

For example,

$$8{,}675.8675 = (8 \times 10^3) + (6 \times 10^2) + (7 \times 10^1) + (5 \times 10^0) + (8 \times 10^{-1}) + (6 \times 10^{-2}) + (7 \times 10^{-3}) + (5 \times 10^{-4})$$

that is, $a_3 = 8$, $a_2 = 6$, $a_1 = 7$, $a_0 = 5$, $a_{-1} = 8$, $a_{-2} = 6$, $a_{-3} = 7$, and $a_{-4} = 5$. Even though we are most familiar with the system for q = ten,

history records ancient civilizations that used q = six, q = twelve, and even q = sixty! For our purposes we are most interested in $q = 2$ (*binary* system), $q = 8$ (*octal* system), and $q = 16$ (*sexadecimal*† system).

First note that for radix q the number system must involve q symbols. For example, for q = ten, the ten symbols are *0, 1, 2, 3, 4, 5, 6, 7, 8,* and *9*; for $q = 2$, the 2 symbols are *0* and *1*; for $q = 16$, the 16 symbols most often used are *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E,* and *F*. Then, if $N = a_n q^n + a_{n-1} q^{n-1} + \cdots + a_2 q^2 + a_1 q^1 + a_0 q^0$, where the a_i are of course chosen from the q symbols, the number is conventionally written by juxtaposition of the coefficients a_i as follows: $N = a_n a_{n-1} \cdots a_2 a_1 a_0$. If $N = a_{-1} q^{-1} + \cdots + a_{-m} q^{-m}$, then the number is usually written by juxtaposition of the a_i preceded by [.]—that is, $.a_{-1} a_{-2} \cdots a_{-m}$. For noninteger $N > 1$ we would have $a_n \cdots a_0 . a_{-1} \cdots a_{-m}$. With this in mind, *counting* in the four different systems mentioned above is accomplished as in Table 2-1.

Conversion from One Number System to Another. Using number systems based on different radices is like talking in different languages. Just as one can translate one language into another, so also one can "convert" from one radix to another. For example, in Table 2-1 numbers on the same row are equivalent—that is,

$$28\ (decimal) = 11100\ (binary) = 34\ (octal) = 1C\ (sexadecimal)$$

The binary, octal, and sexadecimal equivalents of the decimal numbers from 0 to 15 should be memorized.

In order to derive a systematic method for converting from one number system to another, we make use of the well-known result that if N and q are positive integers then there always exist a unique (nonnegative) integer r less than q and a unique integer S such that

$$\frac{N}{q} = S + \frac{r}{q} \qquad 0 \leq r < q$$

Let us apply this to convert an integral number with radix p, say N_p, to its equivalent with radix q, say N_q. In other words, we wish to determine the nonnegative integers $a_0, a_1, \ldots, a_n$, each less than q, such that

$$N_p = a_n q^n + \cdots + a_2 q^2 + a_1 q^1 + a_0 q^0$$

Observe that

$$\frac{N_p}{q} = a_n q^{n-1} + \cdots + a_2 q^1 + a_1 q^0 + \frac{a_0}{q}$$

$$= \underbrace{S_0}_{\text{Integer}} + \underbrace{\frac{r_0}{q}}_{\text{Fraction}}$$

† Also less correctly called *hexadecimal.*

TABLE 2-1. CONVERSION TABLE

Decimal ($q = 10$)	Binary ($q = 2$)	Octal ($q = 8$)	Sexadecimal ($q = 16$)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	*A*
11	1011	13	*B*
12	1100	14	*C*
13	1101	15	*D*
14	1110	16	*E*
15	1111	17	*F*
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17
24	11000	30	18
25	11001	31	19
26	11010	32	1*A*
27	11011	33	1*B*
28	11100	34	1*C*
29	11101	35	1*D*
30	11110	36	1*E*
31	11111	37	1*F*
32	100000	40	20
33	100001	41	21
34	100010	42	22
35	100011	43	23
36	100100	44	24
...	.....	...	...

Since the quotient and remainder *are each unique*, equating the fractional parts we have $a_0 = r_0$ and $S_0 = a_n q^{n-1} + \cdots + a_2 q^1 + a_1 q^0$. Dividing S_0 by q and using our above results, we obtain

$$\frac{S_0}{q} = a_n q^{n-2} + \cdots + a_2 q^0 + \frac{a_1}{q} = S_1 + \frac{r_1}{q}$$

whence $a_1 = r_1$, and so forth.

Hence the systematic conversion method is as follows: Divide N_p by q; the remainder is a_0. Divide the resulting quotient by q; the remainder is a_1. Divide the resulting quotient by q; the remainder is a_2; etc. As an example, we have converted 28 (*decimal*) into binary ($q = 2$) in Table 2-2. Hence,

28 (*decimal*) = 11100 (*binary*)

TABLE 2-2. DECIMAL TO BINARY CONVERSION ($N_p \geq 1$)

	Remainders
2|28	
2|14	$0 = a_0$
2|7	$0 = a_1$
2|3	$1 = a_2$
2|1	$1 = a_3$
0	$1 = a_4$

The work can be more conveniently arranged from *right to left*, as follows:

Divide by 2
←
↓ Place remainder

$$\begin{array}{ccccccccc} 0 & \leftarrow & 1 & \leftarrow & 3 & \leftarrow & 7 & \leftarrow & 14 \leftarrow 28\underline{|2} \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ 1 & & 1 & & 1 & & 0 & & 0 \end{array} \qquad \textit{Remainders}$$

As another example, let us convert 28 (*decimal*) to octal ($q = 8$),

$$\begin{array}{ccc} 0 & \leftarrow & 3 \leftarrow 28\underline{|8} \\ \downarrow & & \downarrow \\ 3 & & 4 \end{array} \qquad \textit{Remainders}$$

whence the converted octal number is 34.

To convert 28 (*decimal*) to sexadecimal ($q = 16$),

$$\begin{array}{ccc} 0 & \leftarrow & 1 \leftarrow 28\underline{|16} \\ 1 & & C \end{array}$$

Hence the converted sexadecimal number is $1C$. [Here, of course, $28 - 16 = 12$ (*decimal*) $= C$ (*sexadecimal*) from Table 2-1.]

Next consider $0 < N_p < 1$. We wish to find nonnegative integer coefficients $a_{-1}, a_{-2}, \ldots, a_{-m}$, each less than q, such that

$$N_p = a_{-1}q^{-1} + a_{-2}q^{-2} + \cdots + a_{-m}q^{-m}$$

Here

$$\begin{aligned} qN_p &= a_{-1} + a_{-2}q^{-1} + \cdots + a_{-m}q^{-m+1} \\ &= u_{-1} + v_{-1} \end{aligned}$$

where u_{-1} is the integral part and v_{-1} is the fractional part of qN_p; hence

$$u_{-1} = a_{-1}$$

and $$v_{-1} = a_{-2}q^{-1} + \cdots + a_{-m}q^{-m+1}$$

Again $$qv_{-1} = a_{-2} + a_{-3}q^{-1} + \cdots + a_{-m}q^{-m+2} = u_{-2} + v_{-2}$$

whence $u_{-2} = a_{-2}$, etc.

The conversion method thus is as follows: Multiply N_p by q; the integral part is a_{-1}. Multiply the resulting fractional part by q; the integral part is a_{-2}; etc.

TABLE 2-3. DECIMAL TO BINARY CONVERSION $(0 < N_p < 1)$

	Integral parts
	0.28
	× 2
$a_{-1} = 0$	.56
	× 2
$a_{-2} = 1$	.12
	× 2
$a_{-3} = 0$	.24
	× 2
$a_{-4} = 0$	.48
	× 2
$a_{-5} = 0$	.96
	× 2
$a_{-6} = 1$	.92
	× 2
$a_{-7} = 1$	.84
	× 2
$a_{-8} = 1$	.68
	....

As an example, we have converted 0.28 (*decimal*) to binary ($q = 2$) in Table 2-3. Hence 0.28 (*decimal*) $= .01000111 \cdots$ (*binary*). The work can be more conveniently arranged from left to right, as follows:

Multiply by 2 → Place integral ↓ part

$2 \times$	0.28 →	.56 →	.12 →	.24 →	.48 →	.96 →	.92 →	.84 →	.68
	↓	↓	↓	↓	↓	↓	↓	↓	↓
Integral parts	0	1	0	0	0	1	1	1	

Note that the process does not necessarily end, as it must for the integral numbers, but may be carried out to any number of significant figures. Conversion to the octal and sexadecimal systems is similarly accomplished.

For nonintegral numbers greater than unity the integral part and the fractional part are done separately as just described.

Note that in converting from binary to octal (or sexadecimal) we divide, or multiply, the binary number by binary 1000 (or 10000), which is equivalent to shifting the fraction point to the left, or to the right, by three (four) binary places. Hence for binary to octal conversion we start at the point and count off the binary bits in groups of three, to the left and to the right. Then we write the octal equivalent for each of these groups of three bits. For example, if the binary number is 11100.110100011, the octal equivalent is

3	4 . 6	4	3
011	100.110	100	011

To convert to sexadecimal, we count off similarly by groups of four. For the same binary number, the sexadecimal number is

1	C . D	1	...
0001	1100.1101	0001	...

(See Table 2-1 for binary-octal and binary-sexadecimal equivalents.) The converse will of course hold.

Converting from Binary to Decimal. Another conversion method, particularly useful for converting from binary to decimal, is based on the following factorization, illustrated for $n = 4$:

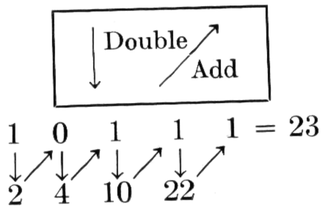
$$\begin{aligned} N &= a_4q^4 + a_3q^3 + a_2q^2 + a_1q^1 + a_0q^0 \\ &= (((a_4q + a_3)q + a_2)q + a_1)q + a_0 \end{aligned}$$

If N is a binary integer, then $q = 2$ and the a's are either 0 or 1, but a_4, the most significant coefficient, must be 1. Hence, to convert N to decimal, start out with 1, and double it, obtaining 2; then add a_3, and double the result; add a_2, and double the result; add a_1, and double

the result; finally add a_0. For example,

$$\begin{array}{ccccc} a_4 & a_3 & a_2 & a_1 & a_0 \\ 1 & 0 & 1 & 1 & 1 \end{array} = (((2 + 0)2 + 1)2 + 1)2 + 1$$
$$= ((4 + 1)2 + 1)2 + 1$$
$$= (10 + 1)2 + 1$$
$$= 22 + 1 = 23$$

The method is to accumulate the result while calculating, and the work from left to right might look like this:



Similarly for N such that

$$N = a_{-1}q^{-1} + a_{-2}q^{-2} + a_{-3}q^{-3} + a_{-4}q^{-4}$$

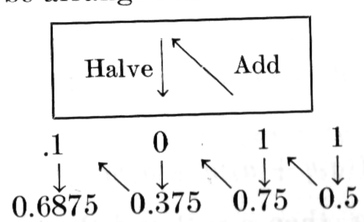
we have the factorization

$$N = \frac{1}{q}\left(a_{-1} + \frac{1}{q}\left(a_{-2} + \frac{1}{q}\left(a_{-3} + \frac{1}{q}\,a_{-4}\right)\right)\right)$$

If N is given in binary, then $q = 2$ and the a's are either 0 or 1. The decimal conversion is calculated by starting with the least significant unit (for example, a_{-4} in our case), working toward the left as follows: Add a_{-3} to .5, and halve the result; add a_{-2}, and halve the result; add a_{-1}, and halve the result. For example,

$$.1011 = \tfrac{1}{2}(1 + \tfrac{1}{2}(0 + \tfrac{1}{2}(1 + 0.5)))$$
$$= \tfrac{1}{2}(1 + \tfrac{1}{2}(0 + 0.75))$$
$$= \tfrac{1}{2}(1 + 0.375) = 0.6875$$

Of course the method is to accumulate while calculating, and the work from *right to left* might be arranged as follows:



This method can be used for octal and sexadecimal conversion to decimal simply by converting the octal or sexadecimal to binary first; e.g., for octal,

53.53 (*octal*) = 101011.101011 (*binary*)

2-3. Number Systems: Arithmetic

Addition and Multiplication Tables. We have already considered the arithmetic operation of *counting* in different number systems. Although all arithmetic can be accomplished by counting, we are more familiar with the laborsaving operations of addition, multiplication, subtraction, and division. Of course these operations may also be carried out in other than the decimal number system. Addition and multiplication are essentially carried out by means of tables. For decimal, binary, and octal the tables are shown in Table 2-4.

TABLE 2-4. ARITHMETIC TABLES

DECIMAL ADDITION TABLE

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

DECIMAL MULTIPLICATION TABLE

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

BINARY ADDITION TABLE

+	0	1
0	0	1
1	1	10

BINARY MULTIPLICATION TABLE

×	0	1
0	0	0
1	0	1

OCTAL ADDITION TABLE

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

OCTAL MULTIPLICATION TABLE

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Arithmetic Operations. In both addition and subtraction the "carry" or "borrow" is handled in other number systems exactly as it is handled in the familiar decimal system. Of course the addition tables are used during the calculation. For example,

1 1 0 1	(*binary*)	7 5 7 (*octal*)
+ , 1 0, 1	(*binary*)	+ , 5, 6 (*octal*)
1 0 0 1 0	(*binary*)	1 0 3 5 (*octal*)
1 1 0 1	(*binary*)	7 5 2 (*octal*)
− , 1, 1 0	(*binary*)	− , 6, 5 (*octal*)
0 1 1 1	(*binary*)	6 6 5 (*octal*)

Multiplication and division in other systems also follow the rules familiar to us in the decimal system, using the appropriate multiplication and addition or subtraction tables during the calculations. For example,

1011	(*binary*)	775	(*octal*)
× 101	(*binary*)	× 56	(*octal*)
1011		5756	
0000		4761	
1011		55566	(*octal*)
110111	(*binary*)		
1011	(*binary*)	775	(*octal*)
101|110111	(*binary*)	56|55566	(*octal*)
101		502	
111		536	
101		502	
101		346	
101		346	

2-4. Sequences of Instructions

We shall initiate our detailed discussion of programming using the *four-address* instruction system. That is, in this section we shall consider instructions which explicitly include four addresses, since a four-address instruction system includes all the essential ideas or basic ingredients of an instruction.

Instruction Format. As we have previously seen, some bits of the instruction are set aside for the operation code designation—i.e., they tell the computer the instruction is add, multiply, divide, etc. The rest of the bits usually denote the four addresses. For the more usual operations that involve two operands, such as addition, multiplication, etc., two of the addresses are the *addresses of the operands.* The third address tells *where the result is to be put;* the fourth address, where to obtain the

next instruction. Hence a typical four-address *instruction format* is as follows (where the four addresses are denoted by α, β, γ, and δ):

Operation code	First operand address α	Second operand address β	Put result into address γ	Address of next instruction δ

For any specific computer using a four-address instruction format, the number of bits in each address and in the operation code must be given, as well as the actual operation code itself. In the design of a computer the number of bits reserved for an address presents an upper limit to the number of words in the addressable memory of the computer. If an address is denoted by n bits, no more than 2^n words can be contained in the directly addressable memory.

Octal Shorthand. In order to illustrate the programming techniques developed in this section, we assume that there are 9 bits in each address ($2^9 = 512$ addressable memory words) and 6 bits in the instruction code, a total of $(4 \times 9) + 6 = 42$ bits in a word. This brings up the first important detail of coding: the actual bits in an instruction are not written out; rather, some shorthand is written instead. For example, in our case the bits would be divided into 14 groups of 3 bits each, and the octal equivalent of each group would be written out. In other words, two octal numbers would represent the instruction, and each address would be represented by three octal numbers. Thus, if 101 011 is the binary code for add, then the instruction that says, "Add the contents of address 011 010 110 to the contents of address 011 100 101, put the result into address 011 110 100, and take the next instruction from 100 000 001," is written in octal notation as

Operation	α	β	γ	δ
53	326	345	364	401

In such cases it evidently facilitates matters to call the addresses in the memory by their *octal numbers.* It will be assumed throughout the rest of this section that such is the case. Also, numerical quantities will be written on the code sheet in *octal* (i.e., they will have to be converted from decimal to octal before being written on the code sheet).

The Computer Manual. For our computer we must have a "computer manual" that gives the operation codes of the different instructions and also precisely defines the meaning of the addresses for each instruction type. For example, see Table 2-5. As can be clearly seen, the coding manual must always be at the coder's side. Two further observations

must be reemphasized before we can proceed with an example. First, *when a word is called into the arithmetic unit from the memory, it is* not *erased from its memory address*, but *remains* there also. Second, *when a word is put into a memory address, it replaces the previous contents of this address*, i.e., it erases what had been there.

TABLE 2-5. OPERATION CODES

Code	Operation	Meaning
53	Add	Add contents of address α to contents of address β; put sum into address γ; take next instruction from address δ
42	Multiply	Multiply contents of address α by contents of address β; put product into address γ; take next instruction from address δ
54	Subtract	Subtract contents of address β from contents of address α; put difference into address γ; take next instruction from address δ
41	Divide	Divide contents of address α by contents of address β; put quotient into address γ; take next instruction from address δ

Example. As we have noted above, a *sequence* of instructions is required for a problem. We shall illustrate such a sequence for a code to evaluate

$$Y = V_{0y}t - \tfrac{1}{2}gt^2$$

where Y is the altitude of a missile projected with an initial Y velocity component of V_{0y} computed at time t. Suppose that V_{0y} = 1,000 fps, g = 32 ft/sec^2, and t = 53 sec. Now 1,000 (*decimal*) = 1750 (*octal*), 32 (*decimal*) = 40 (*octal*), and 53 (*decimal*) = 65 (*octal*). Let us put the numbers

1750 into address 10
40 into address 11
65 into address 12
and 2 into address 13

Let us reserve address 14 for the result of the product $V_{0y}t$, address 15 for t^2, address 16 for gt^2, address 17 for $\frac{1}{2}gt^2$, and finally address 20 for $V_{0y}t - \frac{1}{2}gt^2$. The first instruction would be "Multiply the contents of address 10 by the contents of address 12, and put the result into address 14," where we have yet to indicate where the next instruction will be found. This is written as

Operation	α	β	γ	δ
42	010	012	014	?

Now let us put this instruction into address 1. Next we would write

an instruction to form t^2 and put this instruction into address 2. Now we can finish the first instruction,

42 010 012 014 002

The third instruction would multiply t^2 by g; the fourth instruction would divide gt^2 by 2; and the last instruction would subtract $\frac{1}{2}gt^2$ from $V_{0y}t$. A code is written on a coding sheet, which usually has the left-hand column labeled with addresses and the remaining columns labeled with the parts of an instruction. In this way a clear account is made of exactly which instructions are located in which addresses. On a coding sheet our code would look like Table 2-6. The student should *carefully check* each instruction in this code to be sure that each does what the "remark" says it does. There are several things missing from this code: How do we get the computer started, and how do we stop it when the job is finished?

TABLE 2-6. CODE TO COMPUTE $Y = V_{0y}t - \frac{1}{2}gt^2$

Address	Instruction					Remarks
	Operation	α	β	γ	δ	
000						
001	42	010	012	014	002	Form $V_{0y}t$
002	42	012	012	015	003	Form t^2
003	42	015	011	016	004	Form gt^2
004	41	016	013	017	005	Form $\frac{1}{2}gt^2$
005	54	014	017	020	006	Form Y
006			See text			
007						
010	00	000	000	001	750	V_{0y} constant
011	00	000	000	000	040	g constant
012	00	000	000	000	065	t constant
013	00	000	000	000	002	2 constant
014						$V_{0y}t$ temporary
015						t^2 temporary
016						gt^2 temporary
017						$\frac{1}{2}gt^2$ temporary
020						$Y = V_{0y}t - \frac{1}{2}gt^2$

There are many systems in use for initiating a code on a computer. For example, after the code is read into the computer, the computer might *automatically* take the very first instruction from address 000. In our example we would then have a *dummy instruction*, say "Subtract contents of 007 from contents of 007, put result in 007, and *take next instruction from* 001." We used 007 because our particular code does

not involve 007. In order to stop the computer when the computation has been completed, the last instruction should be one that says simply "Stop the computer." For instance, suppose that 60 is the operation code for such an instruction; then we could put

60 000 000 000 000

in address 006. In such an instruction the addresses α, β, γ, and δ have no use.

The Use of Temporaries. Memory space in a computer is often at a premium and should be conserved. From this point of view the above code is not satisfactory. For note that, after t^2 has been computed, address 015 is not used again. In other words, we could have computed gt^2 and put this result back into 015, and then computed $\frac{1}{2}gt^2$ and put this back into 015 also, eliminating the necessity for using 016 and 017. Similarly, after computing Y, we could put this result into 015, eliminating the use of 020 also.

This brings up an important point about addresses reserved for numbers. There are two kinds of numbers contained in addresses: "permanent," or constant, numbers, such as 2 and g, and "temporary," or transient, numbers, i.e., numbers that appear only in intermediate stages of the computation. These temporary numbers can share their address with other temporary numbers that appear at some other time. Hence we call such memory locations *temporaries*. In our example above, 015 would be such a temporary. The adjusted code becomes as in Table 2-7.

TABLE 2-7. USE OF TEMPORARIES

Address	Instruction					Remarks
	Operation	α	β	γ	δ	
000	54	007	007	007	001	Take first instruction from 001
001	42	010	012	014	002	Form $V_{0y}t$
002	42	012	012	015	003	Form t^2
003	42	015	011	015	004	Form gt^2
004	41	015	013	015	005	Form $\frac{1}{2}gt^2$
005	54	014	015	015	006	Form Y
006	60	000	000	000	000	Stop computer
007						
010	00	000	000	001	750	V_{0y} constant
011	00	000	000	000	040	g constant
012	00	000	000	000	065	t constant
013	00	000	000	000	002	2 constant
014						$V_{0y}t$ temporary
015						Temporary, where final result is found

2-5. Decisions

The Comparison Instruction. We shall next give an example to be coded where a simple decision is made by the computer. Such decisions are usually made by comparison instructions. For example, a comparison instruction might operate thus: "Compare the contents of α with the contents of β; if the contents of address α is greater than the contents of address β, then take the next instruction from address γ; otherwise, take the next instruction from address δ." A few simple conventions can greatly shorten such a statement: Let (α) stand for "the contents of address α"; and let the word *take* mean "take the next instruction from." Then the above statement becomes "If $(\alpha) > (\beta)$, take γ; if $(\alpha) \leq (\beta)$, take δ." Let the operation code for this instruction be 43.

Example. Now consider the code for the following problem: Three numbers, U, V, and W, are found as the contents of addresses 015, 016, and 017, respectively. We wish to write a code that will put the largest of these numbers into address 020. First note that in any case we shall need three instructions that transfer the contents of addresses 015, 016, and 017, respectively, into address 020, and the rest of the code is to decide which of these three instructions to execute. The transfer problem is easily taken care of by adding the contents of the desired address to the constant zero and putting the result into 020. (Hence we need an address whose contents is zero.) Let us put the three such instructions into 005, 006, and 007 (see the code below). To determine which of these to execute, let us first compare $(015) \overset{?}{>} (016)$. If this is the case, then we would compare $(015) \overset{?}{>} (017)$. If this is also the case, then $(015) >$ both (016) and (017), that is, $U > V$ and $U > W$, and hence we would execute (005), i.e., transfer (015) to 020. If $015 \leq (017)$, then $(017) \geq (015) > (016)$, that is, $W \geq U > V$; hence we transfer (017) to 020, i.e., execute (007). If $(015) \leq (016)$, then compare $(016) \overset{?}{>} (017)$. If this is the case, $(016) \geq$ both (015) and (017), that is, $V \geq U$ and $V > W$; hence we transfer (016) to 020, i.e., execute (006). If $(016) \leq (017)$, then $(017) \geq (016) \geq (015)$, that is, $W \geq V \geq U$; hence transfer (017) to 020, i.e., execute (007). The code is shown in Table 2-8.

Several general observations can be made that are exemplified by this code. First, coding is a very exacting technique: every path of every instruction must be followed carefully, for a *single* error invalidates the *entire* code. Second, note that when using a comparison instruction the significance of the $\leq$ must be carefully considered, as well as that of the $>$. Reading a code does *not* teach one how to code—actually writing a code does. Most students will undoubtedly feel uneasy about the above example until they rewrite it, doing the detailed thinking themselves, using the example only as an occasional guide and final check.

The Sign Bit. Up to now we have been making two tacit assumptions, namely, (1) that all numbers are whole numbers (i.e., have no fractional part) and (2) that all numbers and results are positive. Both the assumptions are not necessarily true. They were made in order to

TABLE 2-8. CODE TO DETERMINE LARGEST OF THREE NUMBERS

Address	Instruction					Remarks
	Operation	α	β	γ	δ	
000	54	021	021	021	001	Take first instruction from 001
001	43	015	016	002	003	$U > V$? Yes, try $U > W$; i.e., take 002 No, try $V > W$; i.e., take 003
002	43	015	017	005	007	$U > W$? Yes, U is largest number; i.e., take 005 No, W is largest number; i.e., take 007
003	43	016	017	006	007	$V > W$? Yes, V is largest number; i.e., take 006 No, W is largest number; i.e., take 007
004						
005	53	015	021	020	010	Transfer U to largest number cell
006	53	016	021	020	010	Transfer V to largest number cell
007	53	017	021	020	010	Transfer W to largest number cell
010	60	000	000	000	000	Stop computer
011						
012						
013						
014						
015						U
016						V } three given numbers
017						W
020						Largest number cell
021	00	000	000	000	000	0 constant

concentrate on the coding problems so far considered and avoid unnecessary complication in those discussions. We shall defer full consideration of the first of these problems for a later section, remarking here only that a computer can consider numbers whose magnitudes lie within a restricted range, which depends on the construction of the computer. We have

been illustrating a computer with a range of integers from 0 through $2^{42} - 1$. Hence, before a problem is put on a computer, it must be "scaled" so that all numbers that appear will be whole numbers. Similarly, problems must be scaled to fit on other computers, depending on their ranges. The second problem is solved as follows: A single bit is added on the left to the length of the word, making the word 43 bits long. If this bit is a zero, then the machine is wired to interpret the number as being positive; if this bit is a unit, then the machine will interpret the number as being negative. The circuitry in the arithmetic unit must sense the signs of the operands and compute the proper values for this so-called *sign bit* in the results. The meaning attached to this 43d bit, as far as an instruction is concerned, is different for different computers and is discussed below; it is not a fundamental ingredient of the instruction. On the coding sheet, the sign bit is noted by affixing a + to the left of the number if it is positive or zero, and a — if it is negative.

2-6. Flow Charting and Symbolic Code Aids

A *flow chart* is a diagram, or picture, of a code that is often helpful for visualizing interrelationships between various parts of a code. Such a diagram is almost always made before the specific instructions are written. There are essentially three kinds of symbols used in a flow chart (see Fig. 2-1). The first represents *function calculation;* the second represents *decisions* and the various associated alternatives; and the third,

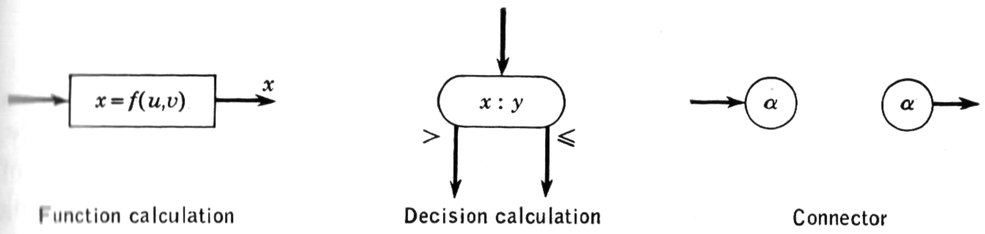


FIG. 2-1. Flow-chart symbols.

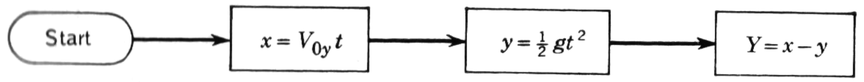


FIG. 2-2. Simple flow chart.

called a (variable) *connector*, is simply a way to eliminate too many crossing lines in the picture or to indicate which lines to follow when one has to continue the diagram on another page.

For illustration, consider the example of Sec. 2-4. The functions $V_{0y}t$ and $\frac{1}{2}gt^2$ are to be calculated, and their difference is the final desired result. Hence we would draw the flow diagram in Fig. 2-2. The first

(left) box represents the instruction in address 001; the second box represents the set of instructions in addresses 002 to 004; the third box represents the instruction in address 005. The arrows indicate the order in which the computer does the indicated computing. Note that the symbols *representing intermediate results appearing on the left-hand side of the equations*, namely, x and y, *appear later in the chart on the right-hand side of an equation.* Only symbols that signify end results, like Y, will not

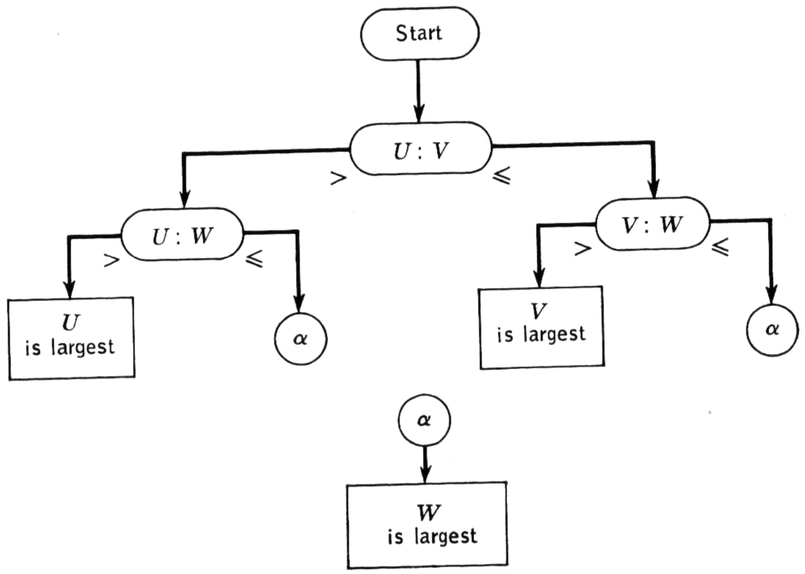


FIG. 2-3. Flow chart including decision boxes.

appear on the right-hand side of an equation. This observation gives a superficial way to check that all intermediate results are used appropriately.

A more complicated flow diagram is associated with the problems of Sec. 2-5. Here decision boxes are used (see Fig. 2-3). The lines leading away from the ovals represent the different paths the computer will take, depending on the outcomes of the decisions. The little symbols ($>$ and $\leq$) near the lines represent the conditions under which the program will go in the corresponding directions. *The proper corresponding condition is indicated when the symbol replaces the colon;* i.e., the computer will follow the left-hand path when $U > V$ and the right-hand path when $U \leq V$. The use of the connector is clear. The top decision box represents (001); on the next lower level the left-hand box represents (002), the right-hand box (003). On the next lower level the left-hand function box represents (005), the right-hand box (006); and the lowest box represents (007).

Symbolic coding is another intermediate aid between the statement of the problem and the final code. Symbolic coding consists in writing a code *not in terms of specific numerical addresses*, but rather in terms of

some name description or other symbolism to represent the addresses. Then, at a later time, specific addresses can be assigned for these symbols, or names, to produce the actual code. The intermediate code in terms of symbols is called the *symbolic code.* This technique is extremely useful, particularly in those cases where one must write instructions involving addresses of constants or of other instructions that have not yet been specifically assigned. For example, this difficulty is encountered in writing the code for the decision example of the previous section, since when

TABLE 2-9. EXAMPLE OF SYMBOLIC CODE

Address	Instruction				
	Operation	α	β	γ	δ
U:V	CMP	U	V	U:W	V:W
U:W	CMP	U	W	ULN	WLN
V:W	CMP	V	W	VLN	WLN
ULN	ADD	U	ϕ	LNC	S
VLN	ADD	V	ϕ	LNC	S
WLN	ADD	W	ϕ	LNC	S
S					
U					
V					
W					
LNC					
ϕ					

writing the instruction for $U:V$, the coder must refer to the not as yet written or located instructions for $U:W$ and $V:W$. A symbolic code for this example might be as shown in Table 2-9. The symbolism of this code is as follows: **U** is the address whose contents is the number U—that is, **(U)** $= U$. **V** is the address such that **(V)** $= V$. **W** is the address such that **(W)** $= W$. **U:V** is the address whose contents has the *compare* instruction concerning U and V, and similarly for **U:W** and **V:W**; **ULN** is the address of the instruction that puts the number U into the largest number cell, and similarly for **VLN** and **WLN**. **LNC** is the address of the largest number cell; ϕ is the address whose contents is the constant zero; and **S** is the address of the stop instruction. **CMP** stands for the *compare* operation code, and **ADD** stands for the *add* instruction code. The assignment of addresses then is shown in Table 2-10.

Steps in Writing a Code. The procedure for programming or coding is as follows: (1) Understand the statement of the problem. (2) Draw a flow diagram for the problem. (3) Code symbolically each box of the

flow diagram. (4) Assign addresses. (5) Write out the final detailed code. The flow diagram presents an *over-all visual picture* of the problem, *organizing the coding effort* so that it can be concentrated on one box at a time. Symbolic coding enables an instruction to be written before

TABLE 2-10. ASSIGNMENT OF ADDRESSES

Symbolic name	*Address (or code)*
U:V	001
U:W	002
V:W	003
ULN	005
VLN	006
WLN	007
S	010
U	015
V	016
W	017
LNC	020
ϕ	021
CMP	43
ADD	53

the coder knows the addresses of constants and other instructions referenced by this instruction. In addition, since the symbolic code names for these addresses are suggestive of their contents, the symbolic code is easier to write and follow.

2-7. Recursion Codes and Instruction Modification

Loops in the Flow Diagram. In Sec. 2-4 we considered the problem of determining the altitude at a time t of a missile projected with an initial y component of velocity V_{0y}. A more realistic problem would be to determine the trajectory of the missile, i.e., the position of the missile, say, after each 10 sec of its flight. For such a problem we use a *loop*, or *recursion code*, i.e., a code which uses a certain set of instructions over again several times. Now at any time t_i, the x and y components of position will be

$$x_i = V_{0x}t_i$$
$$y_i = V_{0y}t_i - \tfrac{1}{2}gt_i^2$$

where V_{0x} is the initial x component of the velocity. To be concrete, suppose that $V_{0x} = 2{,}000$ fps, $V_{0y} = 1{,}000$ fps, and $g = 32$ ft/sec^2. Then at time t_1 (= 10 sec) we would have $x_1 = 20{,}000$ ft and

$$y_1 = 10{,}000 - 1{,}600 = 8{,}400 \text{ ft}$$

At time t_2 (= 20 sec), $x_2 = 2{,}000 \times 20 = 40{,}000$ ft, and

$$y_2 = 1{,}000 \times 20 - 16 \times 20^2 = 13{,}600 \text{ ft}$$

At time t_3 (= 30 sec), $x_3 = 2{,}000 \times 30 = 60{,}000$ ft, and

$$y_3 = 1{,}000 \times 30 - 16 \times 30^2 = 15{,}600 \text{ ft}$$

and so forth. During such a computation it is clear that the same formulas are used over again, each time increasing our t_i by 10 sec. However, the computation should stop when the missile hits the ground, i.e.

TABLE 2-11. COMPUTATION OF TRAJECTORY OF MISSILE

i	t_i	x_i	y_i
1	10	20,000	8,400
2	20	40,000	13,600
3	30	60,000	15,600
4	40	80,000	14,400
5	50	100,000	10,000
6	60	120,000	2,400
7	70	140,000	−8,400

when y_i is no longer positive. In our case it is easily seen that this is when $t = 70$ and $y_7 = 70{,}000 - 16 \times 70^2 = -8{,}400$ ft. Table 2-11 represents the final results, and Fig. 2-4 is the flow chart of the process. In

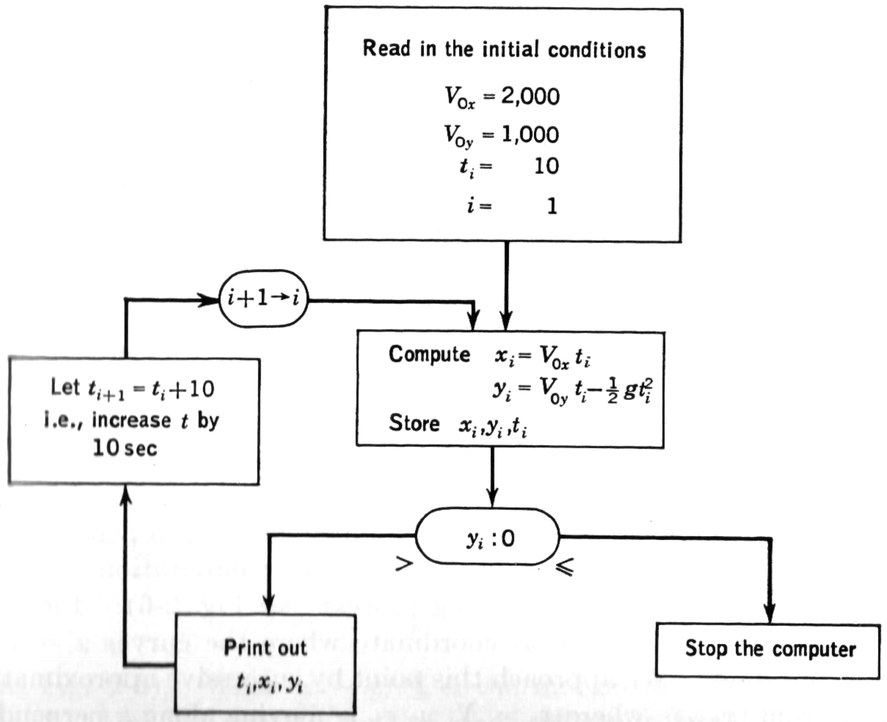


FIG. 2-4. Flow chart for computation of missile trajectory.

the flow chart we have introduced the i notation, where t_{i+1} represents the next time around *and* $i + 1 \rightarrow i$ *means that for the next iteration we replace the old ith values with the new* $(i + 1)$*st values.*

With this illustration it is seen that there are four basic ingredients to a recursion code. It must contain (1) a set of instructions, called the *iteration instructions*, that are to be reused; (2) another set of instructions that modifies the original set each time around; (3) a set of instructions,

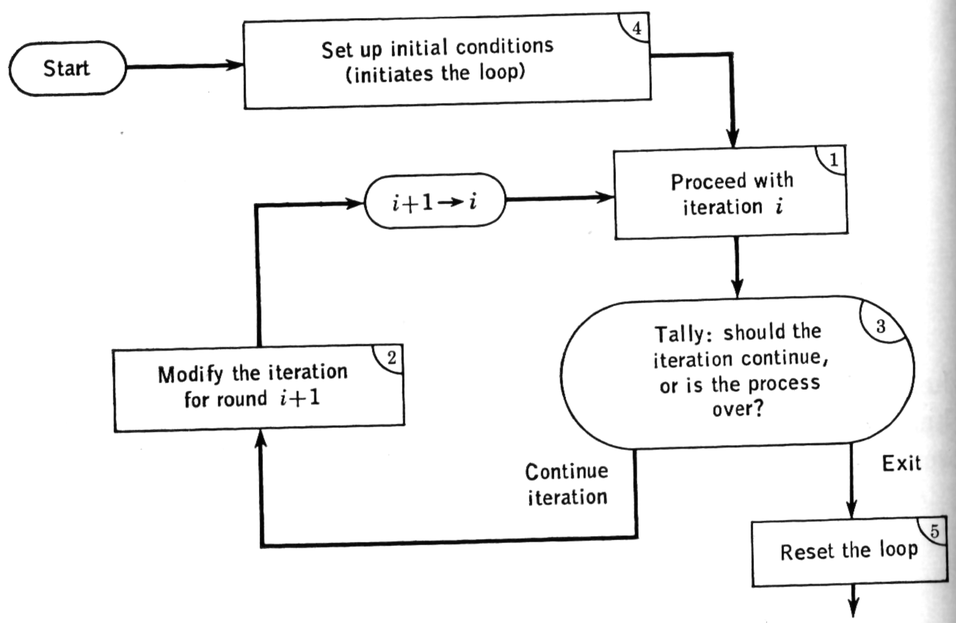


FIG. 2-5. Flow chart of generalized loop.

often called a *tally*, that determines when to *exit*, or break out of the loop, and appropriately notifies the computer; and (4) a set of instructions that sets up the initial conditions and starts the loop. In addition a recursion code often contains (5) a set of instructions that *resets* the loop so that it may be used again by the computer at some future time. A generalized loop can be indicated by the flow diagram of Fig. 2-5.

Sometimes the tally consists of instructions determining whether or not the result of each iteration is smaller than some given number, as occurs often in function computations; or the tally may just count the number of iterations until the desired number have been accomplished. As an example of the former tally, consider the computation of $\sqrt{N}$ to within an error of ϵ by the following process (see Fig. 2-6): First note that $\sqrt{N}$ is the x (and the y) coordinate where the curves $x = y$ and $N = xy$ intersect. We approach this point by successive approximations starting from (x_0,y_0), where $x_0 = N$, $y_0 = 1$, moving along a perpendicular to $x = y$, dropping an altitude to point (x_1,y_1) on $N = xy$, moving

from this point along a perpendicular to $x = y$, dropping an altitude to point (x_2,y_2) on $N = xy$, and so forth. The next point (x_{i+1},y_{i+1}) is found

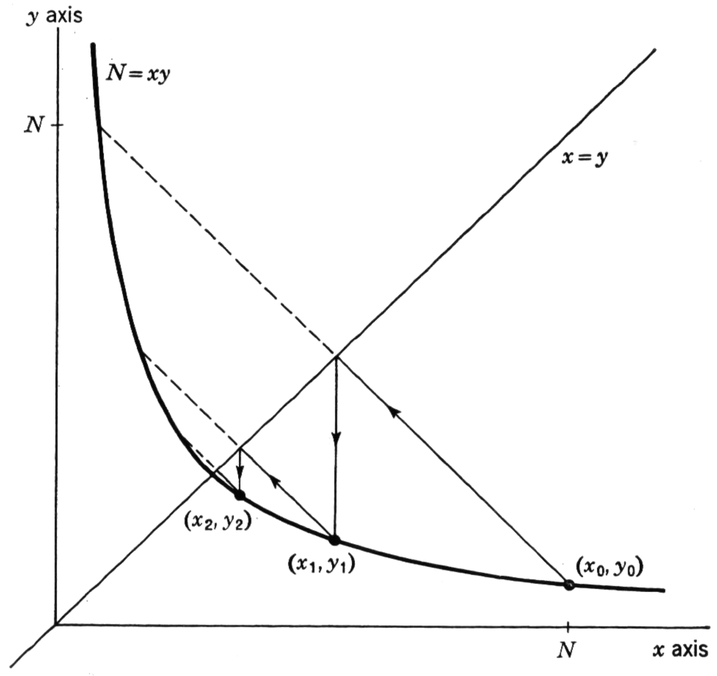


FIG. 2-6. Computation of $\sqrt{N}$.

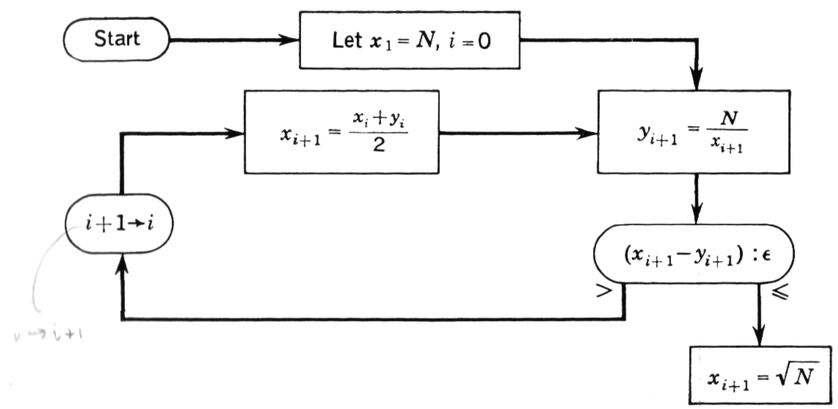


FIG. 2-7. Flow chart for computation of $\sqrt{N}$.

in terms of the present point (x_i,y_i) as follows:

$$x_{i+1} = \frac{x_i + y_i}{2} \qquad y_{i+1} = \frac{N}{x_{i+1}}$$

(see Fig. 2-6). When $x_i - y_i \leq \epsilon$, the desired accuracy has been obtained. The flow diagram for this loop is shown in Fig. 2-7. A symbolic code for this appears in Table 2-12.

As an example of the latter tally, consider a code that computes x^n, where x and n are known. Here, after each multiplication, 1 is added to the tally temporary, and comparison is made with the constant n. A symbolic code for this is given in Table 2-13. Note that initially the

TABLE 2-12. THE SQUARE-ROOT ROUTINE

Address	Operation	α	β	γ	δ	Remarks
000						
001	Divide	N	x	y	002	$y_{i+1} = \dfrac{N}{x_{i+1}}$
002	Subtract	x	y	Δ	003	$x_{i+1} - y_{i+1}$
003	Compare	Δ	ϵ	005	004	$\Delta : \epsilon$
004	Stop					$x = \sqrt{N}$
005	Add	x	y	x	006	$x_{i+1} = \dfrac{x_i + y_i}{2}$
006	Divide	x	2	x	001	
007						N
010						x
011						y
012						ϵ
013						2
014						1
015						0
016						

TABLE 2-13. x^n ROUTINE

Address	Operation	α	β	γ	δ	Remarks
000						
001	Multiply	x	x^n temporary	x^n temporary	002	Form partial product
002	Add	1	Tally temporary	Tally temporary	003	Adjust tally temporary
003	Compare	n	Tally temporary	001	004	
004	Stop					
005						x^n temporary
006						Tally temporary
007						n
010						x
011						1

contents of the x^n temporary must be 1, and the contents of the *tally* temporary must be 0 (why?).

The importance of the recursion code cannot be overemphasized. In fact, as we shall see in the next chapter, special instructions are included in many computers to aid the coding of such codes.

Instruction Modification. One of the fundamental concepts upon which modern computers are based is that of instruction modification, i.e., having a program automatically modify itself. Let us illustrate how instructions can modify instructions by means of the following examples:

Suppose that, during the course of computing for a problem, the computer often needed to know the value of the binomial distribution function $p(x) = \binom{100}{x} (0.1)^x (0.9)^{100-x}$, where $\binom{100}{x}$ is the binomial coefficient, for $x = 1, 2, 3, 4$, or 5. One method for performing such a calculation is simply to write a table into the memory of the computer and "look up" the value each time it is needed.

Suppose such a table was written into addresses 001 to 005; i.e., in address 001 is located $10^4 p(1)$, in address 002 is located $10^4 p(2)$, . . . , in 005 is located $10^4 p(5)$. Note that we must *scale* the values of $p(x)$, since $p(x)$ is less than 1 (for all values of x) but our computer interprets all numbers as greater than 1. Of course the numbers in addresses 001 to 005 will be written in octal.

Suppose also that the value of x (which for this problem can only be 1, 2, 3, 4, or 5) is found in the β position of address 006 and that the result, namely, $10^4 p(x)$, corresponding to this value, is to be put into 007. To perform this task, the code must have an instruction that will transfer the contents of one of the addresses 001, 002, 003, 004, or 005 into 007; such an instruction (where the contents of 010 is the constant zero) is

Operation	α	β	γ	δ	Remarks
Add	010	?	007		Add 0 to ? and put in 007

If the contents of 006 were 1, then 001 should replace the ?; if (006) = 2, then 002 should replace the ?; etc. The problem therefore resolves itself into modifying the instruction with the ? so that ? is replaced by the contents of 006. One way of doing this is to write the instruction first as 53 010 000 007 013, then to add (006) to it, the results being 53 010 001 007 013, and so forth. (Note that we have anticipated placing this instruction in 012 and hence its δ is 013.) But after the code has been used, an additional instruction is needed to reset it. The flow diagram is as in Fig. 2-8. The entire code, then, is as in Table 2-14.

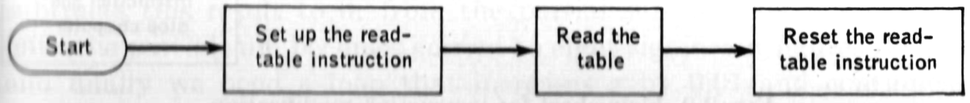


FIG. 2-8. Flow chart of table looking-up routine, indicating *reset* instructions.

TABLE 2-14. BINOMIAL-COEFFICIENT LOOK-UP ROUTINE†

Address	Operation	α	β	γ	δ	Remarks
000	54	010	010	010	011	Take first instruction from 011
001	00	000	000	000	003	$10^4p(1)$
002	00	000	000	000	020	$10^4p(2)$
003	00	000	000	000	073	$10^4p(3)$
004	00	000	000	000	237	$10^4p(4)$
005	00	000	000	000	523	$10^4p(5)$
006	00	000	004	000	000	$x = 4$ (or 1, 2, 3, 5)
007						Result = $10^4p(x)$
010	00	000	000	000	000	0 constant
011	53	006	012	012	012	Set up ? instruction
012	53	010	?	007	013	This is the ? instruction
013	53	010	015	012	014	Reset the ? instruction
014	60	000	000	000	000	Stop
015	53	010	000	007	013	

† See operations defined in Table 2-5, p. 64.

Instruction Modification in Loops. Frequently instruction modification is necessary for the coding of loops. For example, suppose that it is desired to form the sum of a long column of numbers which are located in consecutive addresses of the computer memory from address 030 to 077.

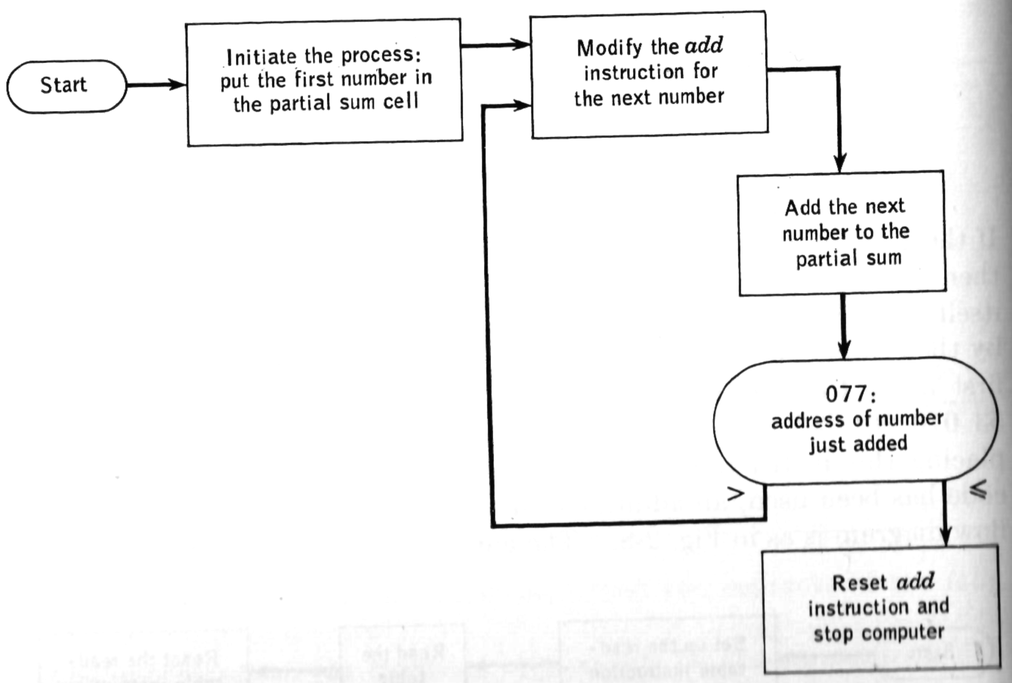


FIG. 2-9. Flow chart for instruction modification.

We could use the same *add* instruction for the successive additions provided that the *add* instruction were modified before each addition so as to add the contents of the *next* successive address to the partial sum each time round. The flow diagram is as in Fig. 2-9 and the code as in Table 2-15. The arrows in Fig. 2-9 indicate the path of the computer through the instructions. The sum is formed in address 001. The student should be sure to understand the purpose of each of the instructions of Table 2-15.

TABLE 2-15. COLUMN-ADDING ROUTINE

Address	Operation	α	β	γ	δ	Remarks
000	54	010	010	010	002	Start at 002
001						Final sum (temporary partial sum)
002	53	010	030	001	003	Put first number into partial sum temporary
003	53	011	004	004	004	Increase β of *add* instruction by 1
004	53	001	030	001	005	*Add* instruction: add next number to partial sum
005	43	012	004	003	006	Tally: is 077 > β of 004? *Yes:* add another number *No:* sum is complete
006	53	010	013	004	007	Reset *add* instruction
007	60	000	000	000	000	Stop
010	00	000	000	000	000	0 constant
011	00	000	001	000	000	1 in β constant
012	53	001	077	001	005	Constant with $\beta = 077$
013	53	001	030	001	005	Constant with $\beta = 030$
030 ... 077						Numbers to be added

Loops within Loops. As an example of multiple loops, or loops within loops, consider the computation of a table of $\sin x$ by means of the infinite series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots$$

for each $\frac{1}{100}$ radian from 0 to $\pi/2$, to eight decimal places. First we would have a loop that formed $x^n/n!$ by multiplying a partial product successively by x/P_i (loop A); then we would need a loop that adds or subtracts this result to or from the partial sum and increases n by 2, until the partial sum becomes correct to eight significant figures (loop B); and finally we need a loop that increases x by 0.01 and continues to

compute the next value of sin x (loop C). The flow diagram for this process appears in Fig. 2-10.

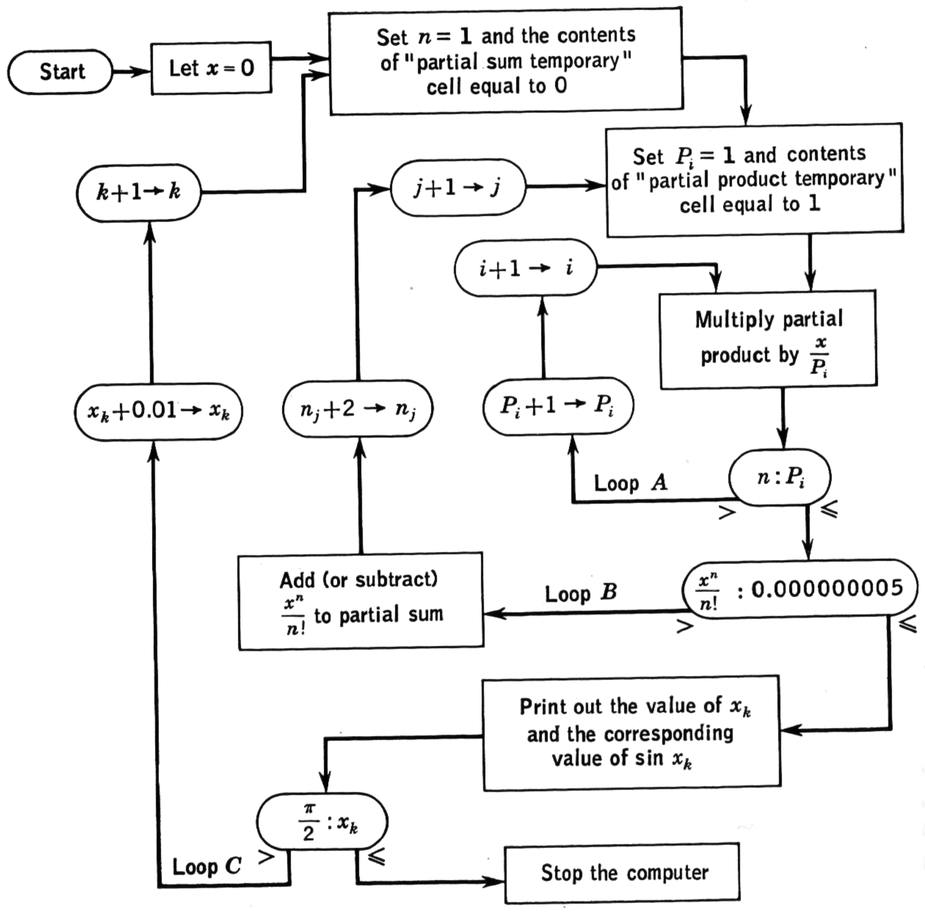


FIG. 2-10. Flow chart of loops within loops. The notation $P_i + 1 \rightarrow P_i$, $n_j + 2 \rightarrow n_j$, and $x_k + 0.01 \rightarrow x_k$, etc., is often used to indicate that in the $(i + 1)$st iteration $P_{i+1} = P_i + 1$, $n_{j+1} = n_j + 2$, and $x_{k+1} = x_k + 0.01$, etc.

2-8. The Subroutine

Use of a Subroutine. A *subroutine* is a subcode that may be used many times during the computation of a program but is written only once in the whole code. As the computer proceeds down the main program, the control will occasionally jump† to this subroutine and then, after doing the subroutine, will jump back to the main program where it left off. This detour from the main program through the subroutine

† The term *jump* is commonly used in the sense of "take the next instruction from." Thus the expression "jump to a subroutine" means "take the next instruction from the address where the first instruction of the subroutine is located." For our present purposes a *jump* instruction usually refers to an instruction whose δ address directs the jump in question. These terms should be clear in context.

may occur several times during the computation of the program. Hence a subroutine must have an *entrance*, a way of getting into it, and an *exit*, a way of getting out of it. Each time an entrance is made to a subroutine, some initial conditions must be set up that are characteristic of the place in the main program from which the entrance was made. For instance, if the subroutine calculates some function, the initial values of the independent variables at that point in the main program must be given to the subroutine. In addition, as an entrance to a subroutine is made, the exit must be set up; i.e., the subroutine must be told *where* to transfer control back to the main program. For each time the subroutine is used, its exit is usually to some different part of the main program. Hence, in order to use a subroutine, the coder must know (1) the entrance, i.e., the address of the first instruction; (2) the addresses of the temporaries in which the initial conditions are to be set up; (3) the addresses of the temporaries whose contents will be the results of the subroutine computation; and (4) the exit, i.e., the address of some jump instruction that is to be preset (when entering the subroutine) so that, when the computation of the subroutine has been completed, the computer will transfer control back to the proper address of the main program.

Consider, for instance, the flow diagram in Fig. 2-11, which illustrates the setting up of the initial conditions and different exits of the subroutine. The A in the circle is a connector and of course means that the program jumps to the subroutine at that point. The B connector is called a *variable connector;* i.e., it indicates that the program jumps either to B_1 or to B_2, depending on how the "variable" connector is set. When we say that $B = B_1$, we mean that the B connector is set so that it will go from B to B_1; and when we say that $B = B_2$, then the B is set to go from B to B_2. The rest of the flow diagram is self-explanatory but should be studied carefully by the student.

Example. As a specific example of the use of subroutines, consider a program which is to compute the radiation intensity S of an antenna at a point in space as a function of the directional angles θ and φ and the radius R given by the equation†

$$S = \frac{30 I_0^2}{2\pi R^2} \left[\frac{\cos\left(\frac{\pi}{2}\cos\theta\right)}{\sin\theta} \frac{\sin\left(\frac{n\pi}{2}\sin\theta\cos\varphi\right)}{\sin\left(\frac{\pi}{2}\sin\theta\cos\varphi\right)} \right]^2$$

where n = a given integer
I_0 = a known constant

† See, for example, J. A. Stratton, "Electromagnetic Theory," p. 451, eq. (53), McGraw-Hill Book Company, Inc., New York, 1941.

Now the sine and cosine functions are evaluated by means of their power series. Since these functions appear several times as functions of different arguments, it would be wise to use their respective power-series codes as subroutines. Figure 2-12 is a flow chart for the calculation of S and indicates the use of the subroutines.

This flow diagram can be used to illustrate some of the details involved in using subroutines. For example, suppose that the first instruction of

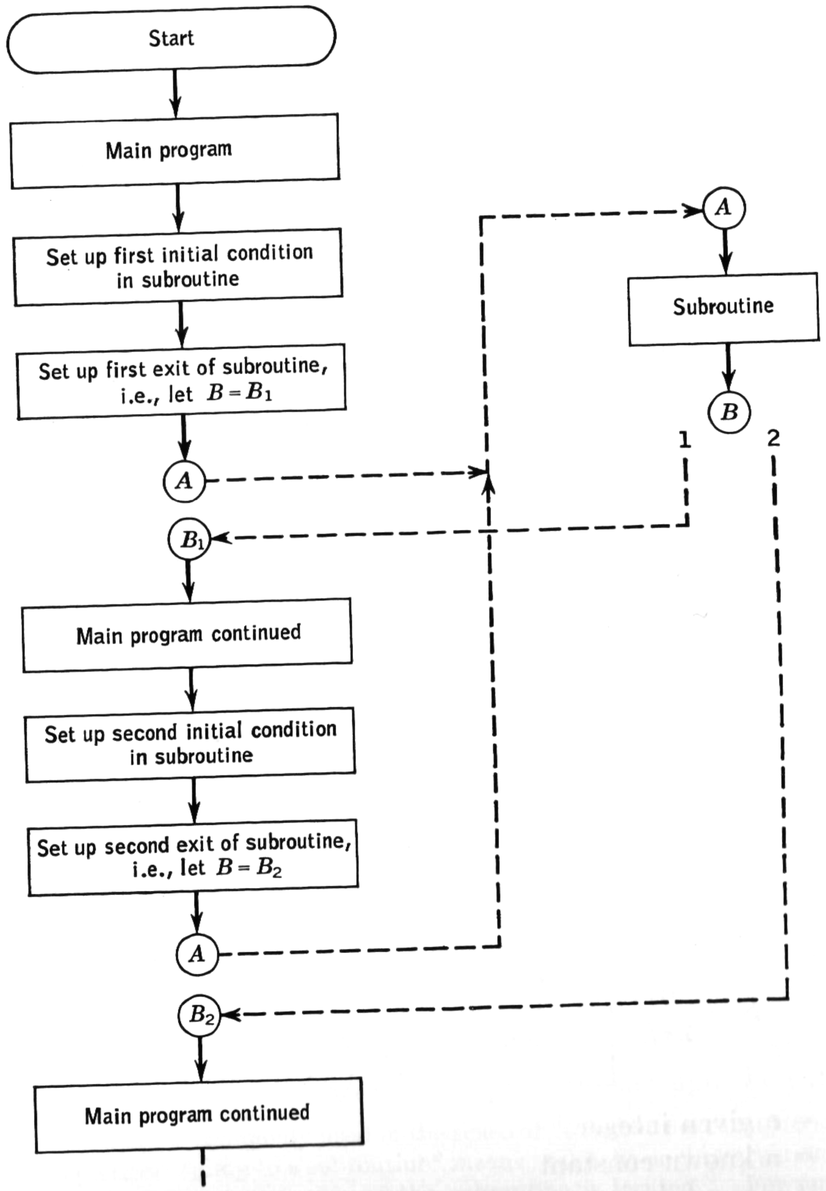


FIG. 2-11. Flow chart for setting up initial conditions and different exits of a subroutine.

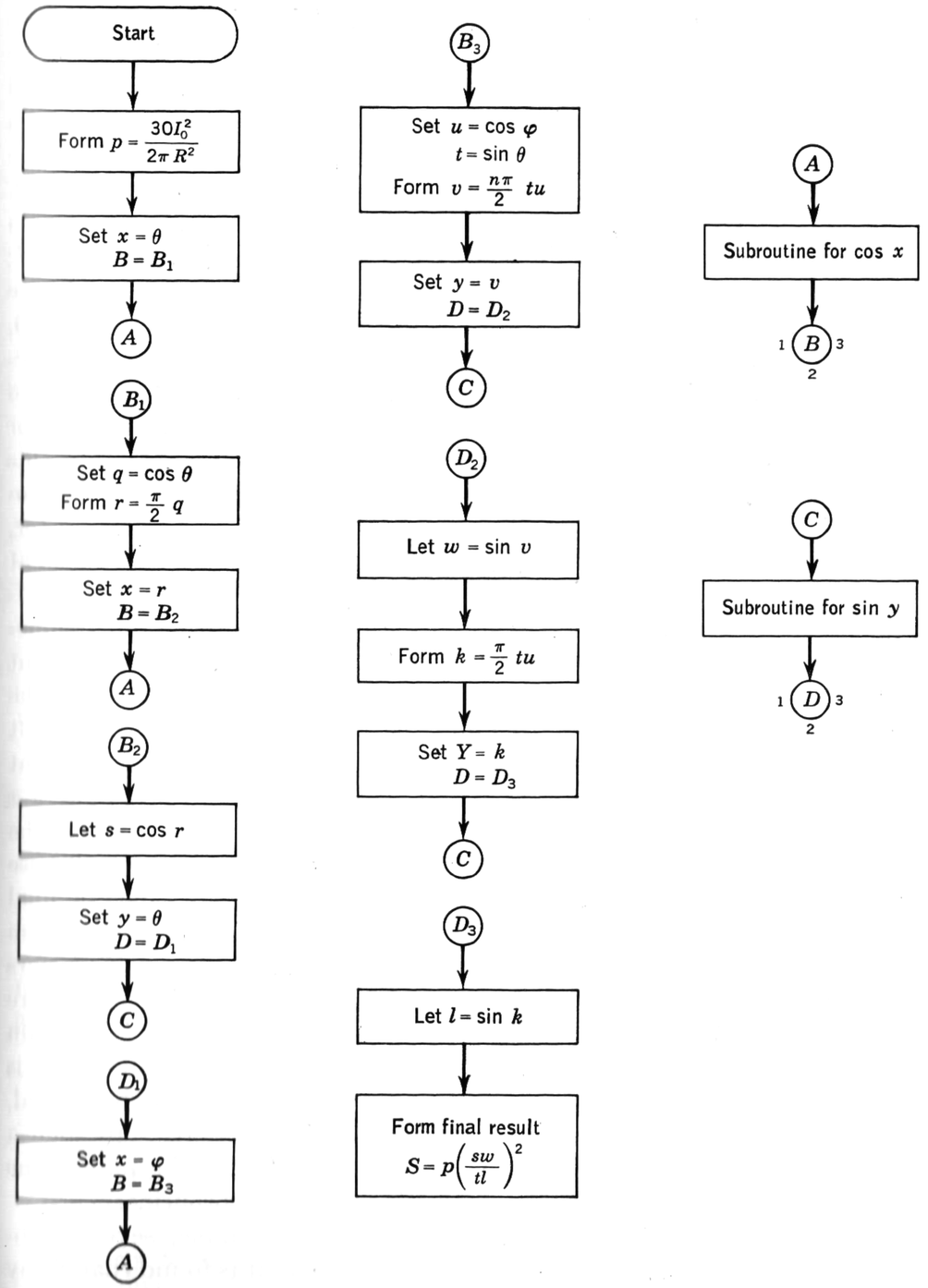


FIG. 2-12. Flow chart for computation of radiation intensity.

the cos x subroutine is located in address 700. Then, in the four-address system, the δ address of the jump instruction to this subroutine would be 700, which completes the first step in using a subroutine. The second step is to set up the exit from the subroutine. One method is to have in address 677 (the address preceding 700) a jump instruction for which the δ address is yet to be designated, i.e., with the δ address initially 000. Suppose that the main program will continue after the subroutine from address 017. Then we would want the δ address of (677) to be 017; then at the end of the subroutine would appear a jump to 677, whence upon executing (677), the computer will jump to 017. To set up (677), the main program would add a constant with $\delta = 017$ and zeros elsewhere to (677). Another method of accomplishing this return jump to the main program is through some sort of special instruction designed for this purpose (see Chap. 3 for further discussion of this point). In this case the jump to the first instruction of the subroutine would automatically memorize its own address in some counter or index register. Then a special jump instruction is placed into address 677 that looks at this counter or register and jumps back to the main program.

For the third step in using a subroutine the main routine must somehow tell the subroutine the values of the initial data, i.e., the independent variables. For this purpose a common agreement is made between the person who writes the subroutine and the user of the subroutine about certain addresses set aside for this purpose. For example, suppose that address 676 were agreed upon to be the address of the initial value of x in the cos x subroutine. Then, before jumping to the subroutine, the main program would transfer the initial value of x into 676 for the use of the subroutine. Similarly an agreement might be made that the final computed value of cos x be placed in 675; when the computer exits from the subroutine back to the main program, the main program first transfers (675) to some temporary that the main program uses. Of course, there is an alternative method for performing these two tasks. The main program could first tell the subroutine where the initial value of x is located and where the final computed value of cos x should be placed, and the subroutine could include the transfer instructions. In such a case the instructions of the main program that "tell" the subroutine would do so by modifying some instructions of the subroutine.

Library of Subroutines. As illustrated by the example, several subroutines may be used in one main program. In fact it is found that many common subroutines are used quite often, such as sin x, cos x, e^x, $\sqrt{x}$, etc. Hence it often becomes profitable to have a *library of subroutines* available to the programmer, stored at all times in some part of the computer memory. There would also be some catalogue kept outside the computer that the programmer can consult when he wants to use a

subroutine. This catalogue would tell where each subroutine is located and all pertinent data about how to use it, such as where to put the initial values of the independent variables, where the computed values of the dependent variables are found, etc. In addition, as we shall discuss in Part 2, automatic-programming aids are devised to make such a library of subroutines easier to use.

2-9. Exercises

Convert the following decimal numbers to binary, octal, and sexadecimal. Carry the conversion far enough so that the converted number differs from the original decimal number by less than $\frac{1}{500}$.

		Answers		
	Decimal	*Binary*	*Octal*	*Sexadecimal*
2-1.	56	111000	70	38
2-2.	0.79	.110010100	.624	.*CA*
2-3.	56.79	111000.110010100	70.624	38.*CA*
2-4.	0.732	.101110110	.566	.*BB*
2-5.	87,231	10101010010111111	252277	154*BF*
2-6.	87,231.732	10101010010111111.101110110	252277.566	154*BF*.*BB*

2-7. Convert -56.79 to binary, octal, and sexadecimal. (HINT: A negative number is converted as if it were positive, and then the negative sign is affixed to the result. Why?)

Convert to decimal:

2-8. 70 (*octal*).
2-9. .624 (*octal*).
2-10. *BB* (*sexadecimal*).
2-11. 70.624 (*octal*).

2-12. Compare the results of Exercise 2-11 with those of Exercise 2-3, and explain
2-13. Make an addition and multiplication table for sexadecimal numbers. [HINT: In the addition table the rows (and columns) count by 1; the row (or column) of the multiplication table counts by n if n is the number of the row (column).]

Perform the following operations in binary:

	Answers
2-14. 111 + 1	1000
2-15. 1010 + 111	10001
2-16. 1110 − 1	1101
2-17. 1 − 1110	−1101
2-18. 1101 × 1011	10001111
2-19. 11011 ÷ 110	100.1

Perform the following operations in octal:

2-20. 6,754 + 777.
2-21. 73 − 56.
2-22. 555 − 62.
2-23. 5,715 ÷ 65.

Using the tables of Exercise 2-13, calculate in the sexadecimal system:

	Answers
2-24. $AFCB + 1AD8$	$CAA3$
2-25. $842D - 65AE$	$1E7F$
2-26. $3C \times D1$	$30FC$
2-27. $1A8E \div 21$	CE

2-28. Using the instructions defined in this chapter, write a code that calculates $x = a^3 - bc$, where $a = 313$ (*decimal*), $b = 983$ (*decimal*), and $c = 838$ (*decimal*).

2-29. In the example given in Sec. 2-4, why are two temporaries needed, that is, 014 and 015? Write a code using only one temporary. [HINT: $V_0t - \frac{1}{2}gt^2 = (V_0 - \frac{1}{2}gt)t$.]

2-30. If an instruction format for a computer allowed 12 bits for each address, what is the maximum number of addressable words possible for the memory of that computer? [*Ans.* $2^{12} = 4{,}096$ (*decimal*) words.] If the first address of this memory is 0000, what is the last memory address? [*Ans.* 7777 (*octal*) (that is, $2^{12} - 1$ written in octal).]

2-31. Using the instructions defined in this chapter, write a code that calculates $X = \text{minimum}(U,V,W)$, where U, V, and W are the contents of addresses 010, 011, and 012, respectively.

Using the instructions defined in this chapter, draw the flow diagram, write a symbolic code, assign addresses, and finally write the final codes to calculate x [where $a = 313$ (*decimal*), $b = 983$ (*decimal*), and $c = 838$ (*decimal*)]:

2-32. $x = a^3 - b \cdot c$.

2-33. $x = b(a + c)$.

2-34. $x = \dfrac{a^3 - b \cdot c}{b(a + c)}$.

If U, V, and W are the contents of addresses 010, 011, and 012, respectively, calculate:

2-35. $x = \text{minimum}(U,V,W)$.

2-36. $x = \text{maximum}(U^2,V^2,W^2)$.

2-37. $x = \text{maximum}(U^3 - VW, V(U + W))$.

HINT: The same symbolic code for Exercises 2-32 and 2-33 appears as parts of the symbolic code for Exercises 2-34 and 2-37.

2-38. Code the flow diagram of Fig. 2-4.

2-39. The symbolic codes of Tables 2-12 and 2-13 are incomplete in that they do not reset the computer for additional runs. Complete these codes accordingly and assign specific addresses.

If x is found in 001 and n is in 002, draw flow diagrams and write codes that will compute the following functions:

2-40. $x^n/n!$. [HINT: Form a partial product by multiplying by $x/(n - i)$.]

2-41. $e^x = 1 + x + x^2/2! + x^3/3! + \cdots$. (HINT: Compute enough terms so that $x^n/n! < \epsilon$; this test will tally the breaking of the loop.)

2-42. The binomial coefficient $n!/x!(n - x)!$.

2-43. The first moment of the binomial expansion, namely,

$$u = \sum_{x=0}^{n} x \frac{n!}{x!(n-x)!} p^x q^{n-x}$$

2-44. Write a routine that forms the sum of the squares of numbers found in addresses 030 to 077.

2-45. Write a routine that puts the numbers found in 030 to 077 in numerical order. HINT: Successively interchange locations of numbers in adjacent addresses if they are not in order; keep track of the number of interchanges each time through; when no interchanges are necessary, then the numbers are in order; that is, for example,

1st time through:	**4, 2, 3, 1**	**2, 4, 3, 1**	**2, 3, 4, 1**	**2, 3, 1, 4**
2d time through:	**2, 3, 1, 4**	**2, 3, 1, 4**	**2, 1, 3, 4**	**2, 1, 3, 4**
3d time through:	**2, 1, 3, 4**	**1, 2, 3, 4**	**1, 2, 3, 4**	**1, 2, 3, 4**
4th time through:	**1, 2, 3, 4**	**1, 2, 3, 4**	**1, 2, 3, 4**	**1, 2, 3, 4**

(no interchanges; so numbers are in proper order).

Construct a library of subroutines as follows:

2-46. Write a subroutine for $\sqrt{x}$ for your library.

2-47. Write a subroutine for $x^m/m!$ for your library.

2-48. Write a subroutine for $\sin x$ for your library. (HINT: This subroutine will itself use the subroutine of Exercise 2-47.)

2-49. Write a subroutine for $\cos x$ for your library. (Do this using only a *single* loop.)

2-50. Write a manual describing how to use your library of subroutines.

2-51. Code the example given on page 81, using your library of subroutines.

CHAPTER 3

INSTRUCTION FORMATS AND OPERATION TYPES

3-1. Three- and Two-address Instruction Systems

Using Fewer than Four Addresses. Up to now we have considered an instruction format containing four addresses. It was remarked in Sec. 2-4 that each of the four addresses is a necessary ingredient to coding. However, four-address systems are not used in practice. Commercially available computers use only three-, two-, or one-address formats, the latter perhaps being the most common.† Computers using fewer than four addresses must compensate by (1) having the omitted information "wired in" and by (2) using more instructions to achieve the same ends. The primary advantage of having fewer than four addresses in the instruction is that for the same word length each address will comprise more bits and hence give the computer a larger addressable memory capability. In the 42-bit format described in Sec. 2-4, 36 bits were reserved for the four addresses, making each address 9 bits long, for a maximum of 512 words in the addressable memory. If three addresses were used, each would comprise 12 bits, allowing a 4,096-word memory; two addresses, 18 bits per instruction, a 262,144-word memory; one address, 36 bits, 68,719,476,736 words. Of course shorter words can be designed for machines with one- or two-address instructions, and this in itself can save a great deal of hardware.

The Three-address Insstrution. The usual three-address instruction is similar to the four-address instruction except that the δ address is omitted from the format. Hence the format becomes

Operation code	Address of first operand, α	Address of second operand, β	Address into which result is put, γ

How does the computer know in what address to find the next instruction? Except for special instructions, the computer will always take as the next instruction the contents of the *next consecutive address* following

† See the Appendix for examples of typical three-, two-, and one-address systems of commercially available computers.

that of the present instruction. That is, the computer will normally take the instructions consecutively in the order in which they are written into the memory. This is accomplished by means of the *current-address register.* This register is first set at the address of the first instruction; then, as each instruction is executed, the contents of this register is normally increased automatically by 1, and this contents is taken by the computer as the address of the next instruction. (Because of this the current-address register is often called the *current-address counter.*) With

TABLE 3-1. THREE-ADDRESS CODE FOR $Y = V_{0y}t - \frac{1}{2}gt^2$

Address	Instruction				Remarks
	Operation	α	β	γ	
0000	53	0007	0007	0007	Take first instruction from 0001
0001	42	0010	0012	0014	Form $V_{0y}t$
0002	42	0012	0012	0015	Form t^2
0003	42	0015	0011	0015	Form gt^2
0004	41	0015	0013	0015	Form $\frac{1}{2}gt^2$
0005	54	0014	0015	0015	Form Y
0006	60	0000	0000	0000	Stop computer
0007					
0010	00	0000	0000	1750	V_{0y}
0011	00	0000	0000	0040	g
0012	00	0000	0000	0065	t
0013	00	0000	0000	0002	2 constant
0014					$V_{0y}t$ temporary
0015					Temporary, where final result is found

this understanding of where the computer will find the next instruction, the only change necessary in the *add, multiply, divide,* and *subtract* instructions given in Sec. 2-4 (see Table 2-5, page 64) is the deletion of the δ address. However, the comparison instruction will need redefining thus: If $(\alpha) > (\beta)$, take next instruction from γ; otherwise take the next consecutive instruction. If $(\alpha) > (\beta)$, then the current-address counter will automatically be set at address γ; if $(\alpha) \leq (\beta)$, the counter will behave as usual, just increasing by 1.

Examples. For instance, the examples coded in Secs. 2-4 and 2-5 would have to be rewritten as shown in Tables 3-1 and 3-2.

There is not much difference between Table 3-1 and its four-address version. Consider, however, Table 3-2. Each branch of this code should be carefully reconstructed by the student so that he may grasp just what problems arise from ordering the instructions and how these

problems are solved. Note that there appear in addresses 0004 and 0001 *dummy comparisons*, which artificially force a jump to address 0012, where the stop instruction is located. Also, a dummy comparison was made in address 0006 in order to reuse (0003). A total of 11 instructions were necessary in the three-address system, while 8 sufficed in the

TABLE 3-2. THREE-ADDRESS CODE TO DETERMINE THE LARGEST OF THREE NUMBERS

Address	Instruction				Remarks
	Operation	α	β	γ	
0000	54	0021	0021	0021	Take first instruction from 0001
0001	43	0015	0016	0005	$U:V$ If $U > V$, take 0005 If $U \leq V$, take 0002
0002	43	0016	0017	0007	$V:W$ If $V > W$, take 0007 If $V \leq W$, take 0003
0003	53	0017	0021	0020	Transfer W to largest number cell
0004	43	0022	0021	0012	Jump to stop
0005	43	0015	0017	0011	$U:W$ If $U > W$, take 0011 If $U \leq W$, take 0006
0006	43	0022	0021	0003	Jump to transfer W instruction
0007	53	0016	0021	0020	Transfer V to largest number cell
0010	43	0022	0021	0012	Jump to stop
0011	53	0015	0021	0020	Transfer U to largest number cell
0012	60	0000	0000	0000	Stop
0013					
0014					
0015					U, V, W given numbers
0016					
0017					
0020					Largest number cell
0021	00	0000	0000	0000	0 constant
0022	00	0000	0000	0001	1 constant

four-address system. The placement of the instructions was somewhat complicated; this difficulty is relieved by *symbolic coding*.

The Two-address Instruction. In the two-address instruction usually both the γ and δ addresses are eliminated. The format is simply

Operation code	Operand address α	Operand address β

The computer finds the location of the next instruction as in the three-

address system, taking care of the function of δ. Where does the result of the operation go? This problem is solved in a two-address system by having the result of every arithmetic operation automatically left in a very special memory location, called the *accumulator*. This special

TABLE 3-3. TWO-ADDRESS INSTRUCTIONS

Code	Operation	Meaning
53	Add	Add (α) to (β), and put result in accumulator
42	Multiply	Multiply (α) by (β), and put result in accumulator
54	Subtract	Subtract (β) from (α), and put result in accumulator
41	Divide	Divide (α) by (β), and put result in accumulator
43	Compare	If (α) > (acc), take β; otherwise go to next consecutive instruction (as usual)
52	Transfer	Transfer (acc) into α, and take β
51	Add and transfer	Add (α) to (acc), and put result into β
40	Multiply and transfer	Multiply (α) by (acc), and put result into β

TABLE 3-4. TWO-ADDRESS VERSION OF CODE OF TABLE 3-1

Address	Instruction			Remarks
	Operation	α	β	
000	54	017	017	Take first instruction from 001
001	42	010	012	Form $V_{0y}t$ in accumulator
002	52	014	003	Put $V_{0y}t$ into 014
003	42	012	012	Form t^2 in accumulator
004	42	011	777	Form gt^2 in accumulator
005	41	777	013	Form $\frac{1}{2}gt^2$ in accumulator
006	54	014	777	Form Y
007	52	015	016	Put Y into 015
010	00	001	750	V_{0y}
011	00	000	040	g
012	00	000	065	t
013	00	000	002	2
014				$V_{0y}t$ temporary
015				Temporary, where result is found
016	60	000	000	Stop computer
017	00	000	000	0 constant

memory cell is located in the arithmetic unit; it has been discussed in Chap. 1 of this book. It suffices to note here that γ is most often understood to be the accumulator, although as we shall see below there can be many variations to this theme. Hence the definitions of our instructions must be rewritten as shown in Table 3-3.

Of course, after the sum is in the accumulator, it must be transferred to an ordinary memory location. Hence we need transfer instructions.

Examples. As mentioned above, the word length is often shortened for two- or one-address systems; so let us, for illustration purposes, use a word length of 24 bits plus one sign bit. In order to increase flexibility,

TABLE 3-5. TWO-ADDRESS VERSION OF CODE OF TABLE 3-2

Address	Instruction			Remarks
	Operation	α	β	
000	54	021	021	Take first instruction from 001
001	53	016	021	Put V into accumulator
002	43	015	005	$U:V$ If $U > V$, take 005 If $U \leq V$, take 003
003	43	017	010	$W:V$ (since V is already in accumulator) If $W > V$, take 010 If $W \leq V$, take 004
004	52	020	012	Transfer V from accumulator into 020
005	53	015	021	Put U into accumulator
006	43	017	010	$W:U$ If $W > U$, take 010 If $W \leq U$, take 007
007	52	020	012	Transfer U from accumulator into 020
010	53	017	021	Transfer W into accumulator
011	52	020	012	Transfer W from accumulator into 020
012	60	000	000	Stop computer
013				
014				
015				U
016				V } given numbers
017				W
020				Largest number cell
021	00	000	000	0 constant

the accumulator is often given an address, say, for example, 777. The example codes as revised are shown as Tables 3-4 and 3-5. Note that two more instructions were needed in Table 3-4 than for the three- or four-address codings. The contents of the accumulator must be carefully watched, or errors might creep in.

In Table 3-5 note how the comparisons were arranged so that the next instruction in sequence could be a transfer instruction direct from the accumulator. This was successfully accomplished for V and U. However, W had to be placed in the accumulator especially for its transfer to the largest number cell. In any event it is to be noted that the placement of instructions and the order of the comparisons become more complicated in a two-address system. Note also that 11 instructions were needed, one more than was absolutely necessary for the three-address system (see Exercise 3-1) and three more than for the four-address system.

3-2. One-address Instruction Systems

The One-address Instruction. In the one-address instruction only one operand address, α, remains. The accumulator does a dual job, standing in for both the second operand address β and the address γ into which the result is put. Address δ is handled as in the three- and two-address

TABLE 3-6. ONE-ADDRESS INSTRUCTIONS

Code	Operation	Meaning
53	Add	Add (α) to (acc); put result in accumulator
42	Multiply	Multiply (α) by (acc); put result in accumulator
54	Subtract	Subtract (α) from (acc); put result in accumulator
41	Divide	Divide (acc) by (α); put result in accumulator
43	Conditional jump	If (acc) is negative, take α; otherwise take next instruction in sequence as usual
52	Transfer	Transfer (acc) into α
51	Replace add	Add (α) to (acc); put result into α
40	Replace multiply	Multiply (α) by (acc); put result into α
44	Jump	Take next instruction from α

systems. There are a large number of variations on this theme, and often additional registers or special memory locations are used. However, in presenting the principles of one-address coding it suffices here to consider only one register, the accumulator. In a one-address system many more kinds of instructions are found essential. For example, in a typical modern one-address computer there are over 200 instructions. This should be compared with the 50 and 40 instructions defined for typical modern two- and three-address computers. It is evident that, the more complex and numerous the instructions that have to be learned by the coder, the harder it is for him to code. In order to help the coder who deals with a machine with, say, 90 instructions, advanced programming techniques are resorted to; these will be discussed below. The necessity for advanced programming techniques to aid the coder prepare programs for a one-address computer system becomes even more striking when one reads the rather tortuous techniques needed to accomplish the simplest of complete operations, such as comparisons, as illustrated by the examples to be presented below. The list of one-address instructions in Table 3-6 is sufficient to meet our needs for the examples.

The transition from two to one address for the *add, multiply, subtract,*

and *divide* instructions is natural enough, but how is comparison performed? To perform a comparison, one of the numbers is put into the accumulator and the other subtracted from it; then a *conditional jump* is used. (A computer with more than one register may compare the contents of two registers.)

TABLE 3-7. ONE-ADDRESS VERSION OF CODE OF TABLE 3-1

Address	Instruction		Remarks
	Operation	α	
000	44	001	Take first instruction from 001
001	42	016	} Clear accumulator, and form t^2
002	53	021	
003	42	021	
004	42	020	Form gt^2
005	41	022	Form $\frac{1}{2}gt^2$
006	52	023	Put $\frac{1}{2}gt^2$ into 023
007	42	016	} Clear accumulator, and form $V_{0y}t$
010	53	017	
011	42	021	
012	54	023	Form $Y = V_{0y}t - \frac{1}{2}gt^2$
013	52	023	Put Y into 023
014	60	000	Stop computer
015			
016	00	000	0 constant
017	01	750	V_{0y}
020	00	040	g
021	00	065	t
022	00	002	2
023			$\frac{1}{2}gt^2$ temporary and result

Examples. Let us shorten our word to 15 bits plus a sign bit, for convenience in this presentation; then the codes for our examples can be written as shown in Tables 3-7 and 3-8.

Note, for the first, that it is necessary to make sure that the contents of the accumulator was zero before using the *add* instruction to transfer a number into the accumulator. Also, we calculate $\frac{1}{2}gt^2$ first so that after forming $V_{0y}t$ in the accumulator we can subtract $\frac{1}{2}gt^2$ directly from the accumulator, thus avoiding an extra clearing, etc., of the accumulator. More instructions are needed than for the other systems: one-address, 12; two-address, 7; three-address, 7; four-address, 7. One less instruction than used will suffice: the student should check this for himself, revising the code to require one less instruction.

There are 25 instructions in the code of the second example as compared

with 11 for the corresponding code in the two-address system, 10 in the three-address system, and 8 in the four-address system. By choosing a more convenient set of one-address instructions (cf. Sec. 3-4) we might have reduced the length of the code somewhat, but not significantly.

TABLE 3-8. ONE-ADDRESS VERSION OF CODE OF TABLE 3-2

Address	Instruction		Remarks
	Operation	α	
000	44	001	Jump to first instruction of code, 001
001	42	035	Clear accumulator } set up comparison
002	53	032	Put V into accumulator
003	54	031	Form $V - U$ in accumulator
004	43	015	If (acc) negative, $U > V$, try $U:W$; if positive, $U \leq V$, try $V:W$
005	42	035	Clear accumulator } set up comparison
006	53	032	Put V into accumulator
007	54	033	Form $V - W$ in accumulator
010	43	025	If (acc) negative, $W > V$, W largest; if positive, $W \leq V$, V largest
011	42	035	Clear accumulator } V is largest number
012	53	032	Put V into accumulator
013	52	034	Put V into largest number cell
014	44	030	Jump to stop
015	42	035	Clear accumulator } set up comparison
016	53	031	Put U into accumulator
017	54	033	Form $U - W$ in accumulator
020	43	025	If (acc) negative, $W > U$, W largest; if positive, $W \leq U$, U largest
021	42	035	Clear accumulator } U is largest number
022	53	031	Put U into accumulator
023	52	034	Put U into largest number cell
024	44	030	Jump to stop
025	42	035	Clear accumulator } W is largest number
026	53	033	Put W into accumulator
027	52	034	Put W into largest number cell
030	60	000	Stop computer
031			U } given numbers
032			V
033			W
034			Largest number cell
035	00	000	0 constant

One-plus-one-address Systems. There is another kind of two-address instruction that deserves mention. It is a two-address instruction where the α and δ addresses are retained, instead of the α and β addresses. In other words, an addition instruction might read: "Add (α) to the accu-

mulator, and take the next instruction from δ." Such a system has the coding characteristics of a one-address system; the purpose of the second address is to allow minimum access coding when a drum is being used. That is, the second address presents some further flexibility in the placing of the instructions in the drum memory. This is often called a *one-plus-one-address system.*

3-3. Decimal Systems

Decimal-coded Binary. The only variations in the coding systems so far described were with respect to the instruction format and, in particular, to the number of addresses that were explicitly displayed in these formats. In all these systems numbers were written in *octal-coded binary;* i.e., a set of octal characters represented a binary number. In fact, the three bits represented by each octal character corresponded to the same number with radix 2: thus, 3 (*octal*) represented 011, 5 (*octal*) represented 101, etc. The instructions were written with the same shorthand, the octal characters representing the real equivalent 3-bit binary numbers.

TABLE 3-9. UNIVAC X-3 CODE

Symbol	Binary code	Symbol	Binary code	Symbol	Binary code	Symbol	Binary code
0	00 0011						
1	00 0100	*A*	01 0100	*J*	10 0100		
2	00 0101	*B*	01 0101	*K*	10 0101	*S*	11 0101
3	00 0110	*C*	01 0110	*L*	10 0110	*T*	11 0110
4	00 0111	*D*	01 0111	*M*	10 0111	*U*	11 0111
5	00 1000	*E*	01 1000	*N*	10 1000	*V*	11 1000
6	00 1001	*F*	01 1001	*O*	10 1001	*W*	11 1001
7	00 1010	*G*	01 1010	*P*	10 1010	*X*	11 1010
8	00 1011	*H*	01 1011	*Q*	10 1011	*Y*	11 1011
9	00 1100	*I*	01 1100	*R*	10 1100	*Z*	11 1100

There is another system for coding numbers and an associated shorthand for representing the bits in an instruction, called *decimal-coded binary.* Either octal-coded binary or decimal-coded binary can be used with four-, three-, two-, and one-address instruction formats.

Usually, in decimal-coded binary, 6 bits are associated with a single decimal number. Since there are 64 different combinations that can be formed with 6 bits, the letters of the alphabet are also assigned binary codes. These assigned codes depend on the logical design of the computer. For example, one such system (UNIVAC) is as shown in Table 3-9.

In such decimal computers a decimal number is written into a word "as is," with no conversion required. Of course the resulting binary number is not a number at all, but a binary code. The arithmetic unit

then handles 6 bits at a time, and is wired so as to produce the correct results. For example, in octal-coded binary we would have

Octal	*Corresponding binary operation as it would appear in computer words*
5	000 101
+ 6	+000 110
13	001 011

In decimal-coded binary we would have

Decimal	
5	000 011 001 000
+ 6	+000 011 001 001
11	000 100 000 100

Note that a word that contains letters *cannot* be added to another word since addition is defined only for numbers. Also, an address must be denoted by pure *numbers,* not letters (for, as we shall see below, we must be able to add to or subtract from the addresses). The letters can be used only for the operation code or for some letter constant.

The advantages of a decimal system are simply that no conversion need be made for numbers and that letters can be read into and out of the computer directly. However, the disadvantage of a decimal system is that it is difficult to write codes where the value of a particular single bit of a word has significance, as is necessary, for instance, in switch sensing; while there are methods for determining the value of some bit of the word, in general the bits are used in the computer in groups of six.

Example. As an illustration of a code in such a system, consider the three-address instructions of Table 3-10, where one decimal symbol is used for the operation code and three decimal digits are used for each address. Here $(\alpha) + (\beta) \rightarrow \gamma$ means "The contents of α are added to the contents of β, and the sum is put into address γ," etc.

TABLE 3-10. THREE-ADDRESS DECIMAL INSTRUCTIONS

Code	Operation	Meaning
A	Add	$(\alpha) + (\beta) \rightarrow \gamma$
M	Multiply	$(\alpha) \cdot (\beta) \rightarrow \gamma$
S	Subtract	$(\alpha) - (\beta) \rightarrow \gamma$
D	Divide	$(\alpha) \div (\beta) \rightarrow \gamma$
C	Compare	If $(\alpha) > (\beta)$, take γ
P	Stop computer	

The code for the first of our examples is shown in Table 3-11.

TABLE 3-11. DECIMAL-CODED BINARY VERSION OF CODE OF TABLE 3-1

Address	Instruction				Remarks
	Operation	α	β	γ	
000	*S*	007	007	007	Take first instruction from 001
001	*M*	008	010	012	Form $V_{0y}t$
002	*M*	010	010	013	Form t^2
003	*M*	013	009	013	Form gt^2
004	*D*	013	011	013	Form $\frac{1}{2}gt^2$
005	*S*	012	013	013	Form Y
006	*P*	000	000	000	Stop computer
007					
008	0	000	001	000	V_{0y}
009	0	000	000	032	g
010	0	000	000	053	t
011	0	000	000	002	2
012					$V_{0y}t$ temporary
013					Temporary where final result is found

3-4. Arithmetic, Logic, Decision, and In-Out Instruction Types

In this and the following section we present many of the important kinds of instructions that are found in modern digital computers. The object is to describe the *concepts* upon which the various kinds of instructions are based, rather than specific details of the instructions themselves. The concepts are chosen from those which are most commonly used. Instructions based on these concepts may take many different forms in the many different computers. For purposes of clarity in this presentation, these instruction concepts have been classified into five major categories: (1) arithmetic instructions; (2) logical and bit-handling instructions; (3) decision instructions; (4) *read* and *write* instructions; (5) recursion-aiding instructions. Although there is some overlapping, it is most convenient to consider each instruction as falling into one, and only one, of these categories. Categories 1 through 4 are considered in this section; category 5 is considered after the discussion of addressing in the next section.

Floating Point. One of the important concepts that appears in arithmetic instructions is the *automatic floating decimal point.* As was seen in some of the above examples and problems, in ordinary arithmetic instructions the coder was limited to handling numbers within the range of the

word length. For example, if the word length is 15 bits plus a sign bit, and if all numbers are interpreted as being nonfractional, that is, if the binary point is considered to be on the right, then only *integers* that lie between 0 and 2^{15}, or -1 and -2^{15}, can be handled by the computer. Obviously there must be some way out of such a totally unacceptable situation. One method is to multiply all numbers involved in a code by an appropriate *scaling factor* so that they fall within the desired range during the computer computations. Of course the results put out by the computer must be readjusted to their actual values. However, it suffices to mention here that scaling presents more work for the coder and is a disadvantage. Many computers avoid this disadvantage by having a so-called *floating point* for arithmetic operations.

In computers with floating-point arithmetic operations, a number is memorized in a word composed of two parts: a *mantissa*, which contains the significant figures of the number, and an *exponent.* For example, one computer that has a 45-bit word uses 36 bits for the significant figures and their sign and 9 bits for the exponent and its sign. Hence this computer can handle numbers whose *absolute values* are as large as $(2^{35} - 1) \times 2^{255}$ (since $2^8 = 256$) or as small as 2^{-255}—that is, numbers within this range with no more than 35 significant bits. A range such as this covers most cases that usually occur and hence relieves the coder of all scaling considerations.

The terms *normalized* and *unnormalized* are often used in connection with floating-point arithmetic operations. These terms are concerned with the problem of determining the appropriate exponent of the result of an operation. The result of an operation is called *normalized* if the exponent is so adjusted that the leftmost bit of the mantissa is a unit bit. This of course maximizes the number of significant figures that can be recorded in a word. A computer operation that presents normalized results is called *normalized;* e.g., we would say *add-normalized.* As an example, suppose that a computer had a 24-bit (8 octal position) mantissa and we multiply-normalized the (octal) numbers (with octal exponent): $.00033333 \times 2^6$ by $.04444444 \times 2^3$. The unnormalized result (double length) is $.0000\ 1753\ 0442\ 4714 \times 2^{11}$. Of course this is a binary number in the computer, and the problem is to determine that exponent which will move the binary number to the left until a unit bit appears in the leftmost position. The number of bit positions the mantissa will have to be moved to the left in our example is $3 \times 4 + 2$ [where 3×4 is the number of bit positions corresponding to four zeros; and since 1 (*octal*) is 001 (*binary*), a move two further positions to the left is necessary]. Figure 3-1 illustrates the binary and octal forms of the double-length mantissa before and after the move.

The final normalized result will therefore be .7654 2212 $\times 2^{-5}$ [where $11 - (3 \times 4 + 2) = -5$]. Of course the computer will do this automatically and will display only the "single-length" result. (See below for further discussion of single- and double-length results of multiplication.) Note that, if the computer binary point is understood to be to the right, the above multiplication would be

$$00033333 \times 2^{-22} \times 04444444 \times 2^{-25} = 00001753 \quad 04424714 \times 2^{-47}$$

If the product were normalized by shifting left, dropping the last eight places would be equivalent to multiplying by $2^{-:0}$ and the exponent would

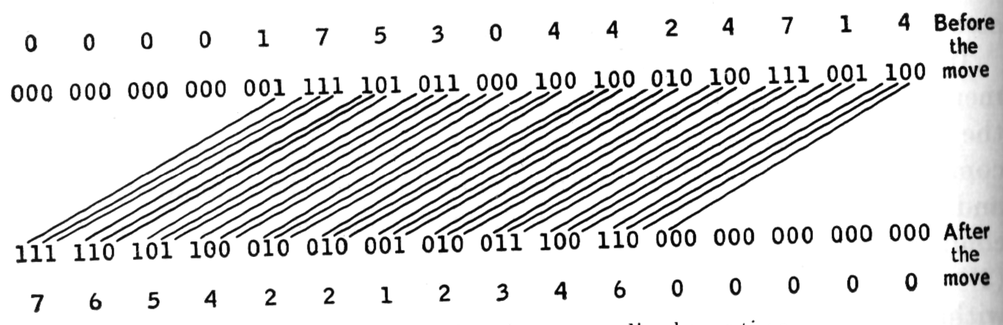


FIG. 3-1. Normalized and unnormalized mantissa.

have to be corrected accordingly: $-47 - [(3 \times 4) + 2] + 30 = -35$, giving 7654 2212 $\times 10^{-35}$ as the result.

When an unnormalized operation is defined, the method for determining the exponent must be clearly stated.

Arithmetic Instruction Types. We can now turn to a description of the kinds of arithmetic instructions. Consider "add" first.

A. ADD. 1. *Ordinary addition* instruction, as described in previous sections.

2. *Floating addition—unnormalized.* The exponent of the result will be that of the largest exponent of the two operands.

3. *Floating addition—normalized.* The exponent of the result will be adjusted so that the most significant bit of the number is a unit.

4. *Absolute-value "add."* In a two-, three-, or four-address system the result would be $= |(\alpha)| + |(\beta)|$.

5. *Add into accumulator.* In a one-address system (acc) + (α) is put into the accumulator.

6. *Add into memory.* In a one-address system (acc) + (α) is put into α.

7. *Add relative.* The instruction code itself tells that the address is to be considered "relative" to some "relative counter" (see Sec. 3-5).

8. *Add with overflow check.* If the sum is greater than the largest number that can be stored in a word, then *overflow* is said to have occurred. If overflow occurs, then the computer is to break the sequence of instructions: e.g., in a three-address system, if no overflow, then take next instruction; if overflow, take γ.

9. *Multiply-add.* In a two-address system, form (α) · (acc) + (β), and put result into accumulator.

10. *Partial add-α address.* In a two or more address system, only the α portion of the words is added. Similarly there are *add-β, -γ address* instructions.

B. SUBTRACT. The kinds of *subtract* instructions are analogous to the kinds of *add* instructions but would also encompass:

11. *Change sign of register.* This is *not* a *subtract* instruction but only changes the sign of the accumulator.

C. MULTIPLY. Before describing the kinds of *multiply* instructions we must first describe briefly how multiplication takes place in an arithmetic unit. First note that, if an n-digit is multiplied by an m-digit number, the result can have as many as $n + m$ digits. In a computer, therefore, the result of a multiplication can contain twice as many bits as appear in a word. Hence there are usually two registers in the arithmetic unit (or one double-word-length register) to record the result of the multiplication. If a word has m bits, then the least significant m bits (which are found in one of the registers, called the *minor register*) are called the *minor multiplication result*, while the rest of the bits are called the *major multiplication result* (and are found in the other register, called the *major register*).

For example, if we consider a 24-bit (8 octal position) word, and multiply 00033333 by 04444444, we obtain as a result 00001753 04424714. Here 00001753 is the major product, and 04424714 is the minor product.

1. *Regular multiplication.* Leaving both major and minor results in the registers.

2. *Major multiplication—unrounded.* The memorized result is the major multiplication result.

3. *Major multiplication—rounded.* The memorized result is the major multiplication result with a least significant unit added to the major result *if* the most significant bit of the minor multiplication result is a unit. If the multiplication is normalized also, the normalization is done first, then the rounding.

4. *Floating multiplication—unnormalized.* The exponents are added, and the mantissas are multiplied.

5. *Floating multiplication—normalized.* After the multiplication the product is shifted and the sum of the exponents adjusted so that the most significant bit is a unit.

6. *Rounding only.* This is *not* a multiplication instruction, but the number in the major register is rounded with respect to the number in the minor register.

D. DIVIDE. In dividing, the quotient appears in the major register, the remainder in the minor register. This brings up some variations in the *divide* instruction.

1. *Quotient divide.* The quotient is the memorized result.
2. *Remainder divide.* The remainder is the memorized result.
3. *Quotient divide—rounded.*
4. *Floating divide—unnormalized.*
5. *Floating divide—normalized.*
6. *Divide with overflow check on quotient.*

Logical and Bit-handling Instruction Types. These instructions are used for two main purposes. First they are used to modify instructions; second they are used when it is desired to find whether the bit in a *Pi* position is zero or a unit, as is often necessary when bits represent switch inputs in a machine.

A. LOGICAL ARITHMETIC. 1. *Logical addition.* There is *no carry*, and the bits in the corresponding positions of the two operands are "added" according to the rule $1 \lfloor + 1 = 1, 1 \lfloor + 0 = 1, 0 \lfloor + 1 = 1, 0 \lfloor + 0 = 0$. For example,

Logical +	0101	A
	0011	B
	0111	$A \lfloor + B$

which can, of course, be generalized to a word of any number of bits in length.

2. *Logical multiplication.* Similar to logical addition except that $1 \lfloor \cdot 1 = 1, 1 \lfloor \cdot 0 = 0, 0 \lfloor \cdot 1 = 0, 0 \lfloor \cdot 0 = 0$. For example,

Logical ×	0101	A
	0011	B
	0001	$A \lfloor \cdot B$

which can be generalized as above.

3. *Logical ring add (or logical inequality).* Similar to logical addition, except that $1 \lfloor \neq 1 = 0$, $1 \lfloor \neq 0 = 1$, $0 \lfloor \neq 1 = 1$, $0 \lfloor \neq 0 = 0$. For example,

Logical ring +	0101	A
	0011	B
	0110	$A \lfloor \neq B$

which can be generalized as above.

4. *Logical compare (or logical equality).* Similar to logical addition, except that $1 \lfloor = 1 = 1$, $0 \lfloor = 0 = 1$, $1 \lfloor = 0 = 0$, $0 \lfloor = 1 = 0$. For example,

Logical compare	0101	A
	0011	B
	1001	$A \lfloor = B$

which can be generalized as above.

5. *Complement.* Units are changed to zeros, and zeros to units; e.g., if $(\alpha) = 0101$, the complement of (α) is 1010.

6. *Extract.* For a four- or three-address instruction, where a bit of (α) is a unit, replace the corresponding bit of (γ) with the corresponding bits of (β). For example,

(α)	000	111	111	000	011
(β)	010	001	101	110	110
(γ)	011	101	110	010	101
Result, found in γ	011	001	101	010	110

7. *Odd-even extractor* (used for decimal computers). Replace those digits of (γ) which correspond (in position) to odd digits of (α) with the corresponding digits of (β). For example,

(α)	222	555	793	246	277
(β)	134	682	304	765	310
(γ)	867	524	116	243	876
Result	867	682	304	243	810

B. SHIFT INSTRUCTIONS. 1. *Right shift.* The whole (α) is shifted to the right by the number of positions specified in the instruction. If the shift is n positions, the effect is the same as multiplying by 2^{-n}. Those bits which run off the right end of the word are lost; those which come into the left end are zeros.

2. *Left shift.* Like the right shift, but in the other direction, i.e., like multiplying by 2^n.

3. *Circular shifts.* Shifts where the bits that go off one end come back at the other as if the ends of the words were connected together. (These are also called "end-around shifts.")

4. *Left significant shift.* Shift till there is a unit in the most significant place. This differs from the above shift instructions in that the number of positions to be shifted is not given by the instruction but is determined during the operation. The number of positions shifted is often recorded in β; the shift is performed in the accumulator for a one-address system.

Decision Instruction Types. When a flow chart of a program comes to a branch point, some form of decision instruction must be used to determine which branch should be taken. As far as the segmented execution of the instructions in a program is concerned, a branch point means that

the program either will continue on normally or will *jump* to another part of the program. The term *jump* is used to indicate that at such a branch point the next instruction is taken from some part of the memory other than the usual sequence.

A. COMPARISON OF WORDS. 1. *Usual comparisons.* If $(\alpha) > (\beta)$, then take γ for four- or three-address computer; if $(\alpha) >$ (acc), take β for two-address computers.

2. *Absolute comparison.* Same as 1, except that the absolute values of the words are compared.

3. *Negative jump.* If (acc) is negative, take α.

4. *Positive jump.* If (acc) is positive, take α.

5. *Zero jump.* If (acc) $= 0$, take α.

6. *Nonzero jump.* If (acc) $\neq 0$, take α.

7. *High jump.* If (acc) $> (\alpha)$, skip next instruction.

8. *Low jump.* If (acc) $< (\alpha)$, skip next instruction.

9. *Equal jump.* If (acc) $= (\alpha)$, skip next instruction.

10. *Least significant jump.* If least significant bit of (acc) is 1, then jump to α.

Of course items 3 to 10 are for one-address systems; in items 7 to 9 the number of instructions skipped may be more than one but would be fixed for a given machine.

B. SENSING OF SIGNAL SWITCHES. 1. *Special signal transfer.* If a certain signal specified by the operation is on, then take α. This instruction is used in conjunction with an instruction that turns on the signal. This latter instruction may be separated in the program from the special-signal-transfer instruction so that the jump occurs some time after the signal was turned on. There may be several signals and associated sets of instructions.

2. *Manual jump.* Here a jump will occur only when a manual external switch has been set before the program reaches the instruction.

3. *Manual halt and transfer.* The computer will stop if the external switch specified by the instruction has been turned on. Then, when the start button is pushed, the jump to α will occur. If the switch is off, the computer jumps without stopping.

4. *Return jump.* In a two-address system the program jumps to (β) for the next instruction; at the same time the contents of the current instruction register (i.e., the address of this particular return-jump instruction) is recorded as (α). The purpose of this instruction is that it is often desirable to jump *back* to the place from which the original program jumped.

C. BREAK POINTS. As was pointed out above, the "sign" bit is an auxiliary bit. The sign bit can be given meaning for instructions as follows: if it is 0, the computer continues to compute as usual; if it is 1, then the computer halts. Important use is made of such a system in detecting errors in a program. Suppose that a coder were running a program on a computer for the first time; he might want to stop the computer at intermediate stages in the program to see whether or not the code were correct up to those points. In order to do this, he would put 1 in the sign bits of those instructions at which he wants to stop the computer so that he could examine the partial results. An alternative to this procedure, which serves the same purpose, is to have the computer take the next instruction from a standard, predetermined location in the memory whenever a 1 appears in the sign bit of an instruction. At the standard position a special code might be located that can do various things to aid the coder detect errors and correct his code (see Sec. 5-2).

Read and Write Instruction Types. These instructions operate the input and output electromechanical units.

A. MAGNETIC TAPES. These instructions can enable the computer to move the magnetic tapes forward or backward, or completely to rewind them. They enable the computer to read from the magnetic tapes into the higher-speed memory. The number of words written onto or read from the magnetic tape is specified either in terms of individual words or, more usually, in terms of the number of *blocks of words*, where by a block of words we mean some predetermined number of words. If, for example, there are eight words in a block, then only multiples of eight words can be handled. Similarly, in just moving the magnetic tapes, the distance to be moved is specified in either single words or blocks of words. The instruction code tells whether to read or write, or move the tapes backward or forward; the number α tells how many words or blocks are to be considered. For reading and writing, the number β is the address that the first word is read into or written out of.

B. PUNCHED TAPES AND CARDS. Instructions that tell the machine to read in from paper tape or to punch paper tape either give the initial and final addresses to be read into or out of or else give the first address and the number of words to be considered. Sometimes in reading-in paper tape, only the initial address is given, and a signal is put on the paper tape to tell the computer when to stop reading. Punched-card input-output is similar.

C. DISPLAY. For displays on an oscilloscope each instruction may tell the display to exhibit one dot or other geometrical pattern, and the instruction gives the coordinates of that dot or of some point on the pattern.

D. CONCURRENT IN-OUT. Some computers have concurrent input and output. In these cases the computer can compute at the same time that words are being read into or out of the computer.

In addition it should be mentioned that some computers have such

complicated methods of reading in and out that special codes must be written to facilitate these processes, and these codes must be available at all times inside the computer's memory. Often these codes must be put into the computer at first by hand methods, in a sort of bootstrap operation. Once the routines are in the computer, the operation may never need to be repeated. However, these methods are so peculiar to the particular computer involved that it is not worthwhile considering them here.

In the Appendix we have summarized some of the instructions on three typical, commercially available computers which employ many of the concepts discussed in this section.

3-5. Indirect Addressing and Relative Indexing

Instructions can be defined to perform many noncomputational functions for the purposes of extending the computer's capabilities and

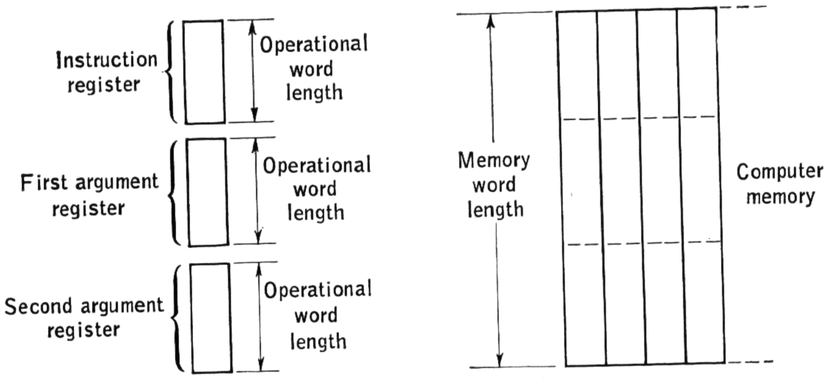


FIG. 3-2. Illustrating "memory-word" length being a multiple of the "operational word" length. Hence in this case a single memory-word access produces three operational words for the computation unit.

enhancing the programmer's convenience. For the former purposes, variable word addressing and indirect addressing are utilized; for the latter purposes, relative indexing and special recursion-aiding instructions are utilized. In this section we consider the principles upon which such instructions are based.

Variable Word Addressing. In currently available computers, the memory access time is generally longer than the instruction execution time. To take fuller advantage of the high computational speed offered by modern computers, it therefore often becomes desirable to select several words from the memory simultaneously, or at least *during the same phase.* The words thus chosen are read out of the computer memory into high-speed registers available in the computing unit. One technique for accomplishing this is to have one "memory word" really equivalent to several "operational words" (see Fig. 3-2). Thus, for example, if

three operational words are equal to one memory word, the instruction and its two arguments could be all read into the computing unit at once. For more flexibility, the number of operational words to be accessed at the same time can be put under the control of the programmer. In effect, we are then fixing the length of the operational word and allowing the memory-word length to vary.

A closely allied capability is to place the *length* of the word to be accessed under the control of the programmer. In this way, variable-length business records, files, and other items can be more efficiently

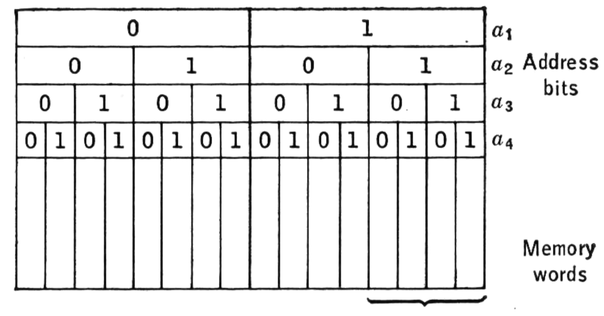


FIG. 3-3. Illustrating partial addressing. If $n = 4$ and the full address is $a_1a_2a_3a_4$, then the partial address for $a_1 = 1$, $a_2 = 1$ would access the four words of addresses 1100, 1101, 1110, 1111, shown bracketed.

organized in the memory. Here the length of some component of the operational word, say a character, is fixed, and the number of such characters is allowed to vary, resulting in variable-length words.

A. PARTIAL ADDRESSING. Varying the number of words to be accessed in the memory can be accomplished by using a shortened address. Thus, consider a 2^n-word memory, where each word is identified by an n-bit address: an $(n - 1)$-bit address will identify one of $2^{n-1} = 2^n/2$ pairs of words; or in general a partial $(n - p)$-bit address will identify one of $2^{n-p} = 2^n/2^p$ groups of 2^p words. For illustration, if $n = 4$ (see Fig. 3-3), then the partial address 11, say, would access the four adjacent addresses

1100 1101 1110 1111

Partial addressing can be incorporated into the instruction format of a decimal computer by filling in the blank part of the partial address with a letter character. For a binary computer, a convention might be made that every address is to have a unit as its most significant bit; then the displacement of this unit to the right would determine the number of bits omitted in the partial address.

B. OVERADDRESSING. Varying the length of the accessed word involves the addressing of a component of a word, say a 6-bit character. A special

character is utilized to define the end of a word. Then to address a word, only the address of its first character is used; as the word is read into the computing unit sequentially, one character at a time, the end-of-word character is sensed, stopping the memory access process (see Fig. 3-4). Since each character is being addressed, the number of bits in the address will be more than is required by the number of words that could be formed in the memory (except in the case where all the words are chosen to be one character long)—hence the term *overaddressing*. This method truly represents a variable word length, but the price paid is an address with sufficient bits to address the shortest possible word. Also, special instructions must be made available to automatically lengthen, shorten,

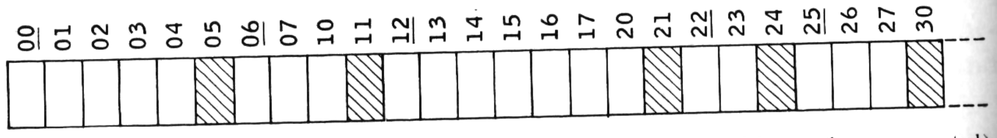


FIG. 3-4. Illustrating overaddressing. The figure shows the address (in, say, octal) of each character in the memory, where the crosshatched characters are end-of-word characters. The address of the first word is 00, and it contains five characters; the address of the second word is 06, and it contains three characters; the address of the third word is 12, and it contains seven characters; and so forth.

or initiate words by moving, omitting, or inserting end-of-word characters at specified locations. There are many variations as well as combinations of these methods. For example, by shortening the component or character address, we might make the computer read larger (or multicharacter) components from the memory, until the end-of-word signal character appears.

Indirect and Augmented Addressing. For very large memories, the number of bits required for an address may be more than the number relegated to the address part of an instruction. In such a case, the large memory can be accommodated by having the instruction directly address *not* the desired operand, but rather another word whose entire contents will be the address of the operand. Generally the "other word" is a high-speed register reserved for the purpose. If there are 2^r such registers, then r bits are required to address one of them; this is called *proper indirect addressing.* A similar technique is to have the instruction indicate *only part* of the address of the desired operand, and also the address of a register the contents of which will *complete* the desired address. This *augmented addressing* can be done on either the most significant or the least significant end of the indicated address. In either case, the *most significant part* of the resulting address is interpreted as denoting the part, or *area*, of the memory to be accessed, and the *least significant part* gives the address of the desired word *within that area.* If there are 2^r high-speed registers, each of contents R bits, 2^m memory words, and i bits

in the instruction address, then in the former case we would need $i \geq r$, and $R \geq m$ for addressing "gain," and in the latter case would need r bits in the instruction for addressing the register, where $r + i < R + i$ and $R + i \geq m$.

A. INDIRECT ADDRESSING. Consider as an example a three-address instruction format with a 27-bit (9 octal position) word. Suppose three octal positions are used for the operation, leaving six for the three addresses, or two octal positions per address. This will enable a maximum-sized memory of 2^6, or 64, words. However, if it is desired to extend the memory size, indirect addressing must be utilized. Suppose

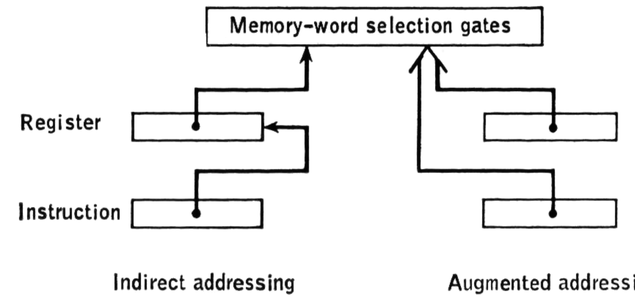


FIG. 3-5. Diagrammatic illustration of indirect and augmented addressing. An arrow arising from inside a rectangle indicates that its contents is utilized; an arrow directed to the side of a rectangle indicates the designation of the address of the register.

it were desired to have 64,000 words for the memory, requiring a 16-bit address. To accomplish this, the instruction addresses could refer to one of up to 64 indirect-address registers, each of which itself consists of 16 bits. Then, previous to the execution of the instruction, the appropriate registers would be loaded with the desired 16-bit addresses. When being executed, the instruction would refer to the addresses of the registers whose contents are the desired memory addresses; the computer would automatically access these desired memory addresses (see Fig. 3-5).

B. AUGMENTED ADDRESSING. Consider as an example a one-address instruction format with an 18-bit (6 octal position) word. Suppose that two octal positions are used for the operation, leaving only four for the address, and that 250,000 words comprise the total memory. Augmented addressing would then be required. An augmenting register of 6 bits would be preloaded, and the desired address would be interpreted by the computer as being the 6 bits of the register taken together, or augmented, with the 12 bits of the instruction. If the register address bits are considered to be the most significant bits, then the contents of the register can be thought of as denoting which of 64 possible memory banks is being utilized, each such memory bank containing 4,096 words (see Fig. 3-5). Such a system might be used, for instance, in conjunction with an overaddressing technique, as described above.

Relative and Compound Addressing. The previously discussed methods of this section had as their aim the *extension of the computer's capabilities.* We now turn to techniques designed to aid the programmer. Relative and compound addressing are primarily *aids to instruction modification.*

A. RELATIVE ADDRESSING. Relative addresses are associated with so-called *index registers.* An index register is a very special memory cell or register that can contain only a single address; i.e., it is a "short" memory cell. The computer must have special instructions that can put a number into such a register, add to the number in the register, etc.

An address appearing in an instruction is said to be interpreted "relative" to a register if the computer considers, instead of the contents of the address indicated, the contents of the word whose address is the *sum* of the index-register *contents* and the *indicated address.* For example, if the contents of the index register were 002 and the address in the instruction were 053, then the computer would consider the contents of

$$002 + 053 = 055$$

In cases where many index registers are available, the computer instruction word will contain, in addition to the indicated address, the *address of the index register* relative to which the indicated address is to be considered (see Fig. 3-6).

The use of such relative addresses is to aid the coding of iterations, or loops. As illustrated by some of the problems of the previous chapter, each time around a loop it is often necessary to increase some addresses of the main set of instructions by 1 (or some other integer). In such cases a set of instructions modifying the instruction must precede the main set of instructions. But with relative addresses all that need be done is to increase the index-register contents by 1 (or whatever is the necessary integer for a particular case) by means of a "register-setting" instruction. One variation of this system is to have several such index registers, which can be used in many different ways. For example, an instruction preceding the use of relative addresses can determine to which register the addresses are to be considered relative. Or the addresses may be considered relative to the sum or some other function of the numbers in the registers.

Relative addressing can be considered as the fundamental concept of this section. Then augmented addressing can be interpreted as having some positions in the index register correspond to zeros or unrepresented bits in the direct address of the instruction itself. Finally, indirect addressing can be thought of as being the extreme case, in which the direct address is null, i.e., all zeros, and does not actually appear.

B. COMPOUND ADDRESSING. For greater flexibility, indirect addressing, augmented addressing, and relative addressing can be combined.

For example, the indirect address could be augmented, which would effectively enable the combining of two registers. Or an indirect address could be made relative to the contents of an index register. Another combination could be to have the instruction address a register for the basic indirect address, and also to address two index registers for relative addresses. The desired address then becomes the sum of the indirect address and the contents of each index register. By incrementing the index registers, a doubly indexed array can easily be stepped through. Of course, many such combinations can be made. Finally, each memory

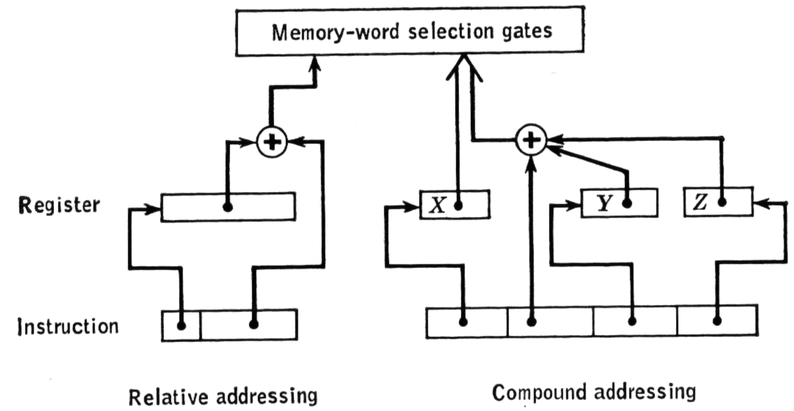


FIG. 3-6. Diagrammatic illustration of relative and compound addressing. An arrow arising from inside a rectangle indicates that its contents is utilized; an arrow directed to the side of a rectangle indicates the designation of the address of the register. In the compound-addressing case, the use of register X may be considered augmented addressing, for choosing the proper memory bank, while the use of Y and Z may be considered relative addressing for double-indexing purposes.

word could be made available as an index register. Here we could have an address in an instruction word refer to the address of another word whose contents will contain the address of an index register (see Fig. 3-6).

The use of such index registers is associated with four functions: (1) address modification, (2) incrementing the index register, (3) index or tally testing, and (4) the initial setting of the contents of the index register, tally, and increment. The address modification is the forming of the desired address as the sum of the index-register contents and the indicated address; the incrementing of the index-register contents is frequently accomplished by utilizing a counter for the register itself. The use of lists and loops requires a tally that signifies the end of the list and the exit for the loop; the tally condition is determined by a comparison between the contents of the index register and some tally constant, which may be zero, some number in the instruction, or a number in some other word. Special instructions must be defined to set the initial contents of the index register, to specify the increment when it is a variable, and so forth. These will now be discussed.

Recursion, or Loop-aiding, Instructions. These instructions are designed to aid in the programming of loops. In the examples and exercises given in this and the previous chapter, each time around the loop the main set of instructions usually needed to be modified by means of additional instructions. As was seen above, index registers can aid in eliminating these additional instructions. Also, loop codes require that a tally be kept to allow the computer to jump out of the loop. The purpose of recursion, or loop-aiding, instructions is to facilitate the use of the index registers and to aid in keeping the tally automatically.

A. INDEX-REGISTER MODIFIER. Adds 1 to the index register and jumps to α. Here the loop tally is not considered.

B. FILE AND MODIFY INDEX REGISTER. This instruction stores the contents of one of the index registers (assuming that there are several such registers) as (α); then another register designated by the instruction is set to the number β (that is, *not* the contents of β, but β itself), and this register is then considered the one to be used until another such *file* instruction occurs. Since resetting and going to another register essentially constitutes a jump, this instruction can act as a return jump. However, if the number β of the *file* instruction itself is considered relative to the same register that the *file* instruction is setting, then this register is set to its original reading plus β; that is, the register is increased by β. As we have seen in the index-type instructions, resetting a register is an aid to recursion codes; however, *no* tally is considered.

C. TALLY JUMP. Subtracts 1 from the number α of the instruction; if the result (that is, $\alpha - 1$) is positive, replaces the number α by the number $\alpha - 1$ (in the α-address position) of the instruction, and jumps to take as the next instruction the contents of β; otherwise the program continues sequentially as usual. This instruction does not consider the index registers.

D. MODIFY INDEX REGISTER AND TALLY. Compares the contents of the index register with the number α. If the contents of the index register is greater than the number α, then the number α is subtracted from the index register and the difference is placed in the index register. Then a jump is made to β. Otherwise the program continues in sequence. Note that the index-register contents acts both as tally *and* as a relative address. Of course this must be used in conjunction with an instruction that initially sets the contents of the index register.

In the Appendix we have summarized the use of indirect addressing and relative indexing in some commercially available computers.

3-6. Special Coding Techniques

Multiway Branch. Our instruction systems described above allow for only a two-way branch in a code. That is, by means of a comparison

instruction, the computer can determine along which of two possible directions to continue in the code. However, it often occurs that a many-way branch is needed in a code; this must be programmed. Just as the decision process for a two-way branch was reduced to a comparison of numbers, so a many-way branch may be reduced to comparisons of numbers. For example, if a code is to have a four-way branch, then, to determine which branch is taken in any case, the code might determine a number N which can take only the values 1, 2, 3, and 4. Thus one way of coding the four-way branch would be a *split tree* of successive comparisons, as in Fig. 3-7.

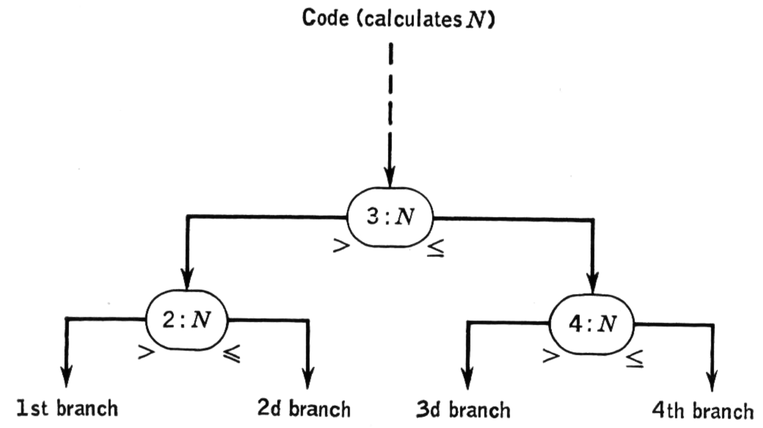
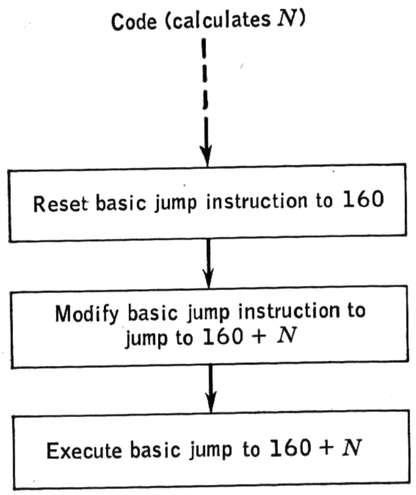


FIG. 3-7. Split tree of comparisons for multiway branch.

There is another, more efficient method for coding a many-way branch, using a so-called *jump table.* Here the number N that decides which branch the computer should take is used as an instruction modifier (see Fig. 3-8). The basic *jump* instruction is modified so that, when it is executed, the computer will jump to the appropriate branch jump instruction of the jump table. This branch jump then jumps to the proper branch of the code.

Switch Sensing. In real-time applications of digital computers (see Sec. 1-3), it is often convenient to connect the outside world to the computer by means of a battery of switches. These switches are attached to a very specially built word in the computer's memory, and each switch controls a single bit of this word. If a switch is on, then the computer reads a unit in the position of the word corresponding to this switch; if the switch is off, then the computer reads a zero. Thus at any time this special word gives the status of the corresponding battery of switches. Let us in addition assume that this special word behaves the same as any other word of the computer's memory, except that the computer *cannot* read *into* it.

There are two ways of interpreting these switches, called *on-off* switch sensing and *one-shot* switch sensing. For on-off switch sensing the computer is to determine for each switch whether it is on or off. The nth switch can be sensed in this way as follows: Shift the switch word left until the nth bit is in the most significant place; then compare this shifted word with a word of all units except in the most significant place. For example, consider a one-address system with the following *shift* instruction: "Shift (acc) left the number of positions α." Thus suppose that the switch word were 011 110 101 110 001, and that we wanted to determine whether $P8$ (the eighth bit from the right) were 0 or 1. We would place this word into the accumulator and perform: Shift left 7— i.e., shift the word left seven positions. For the comparison, a negative jump would be used after subtracting (acc) from 011 111 111 111 111. If the result is negative, then the shifted word had a unit in the most significant position, that is, $P8 = 1$; if the result is zero or positive, $P8 = 0$.



The jump table

Address	Branch Jump
161	Jump to 1st branch
162	Jump to 2d branch
163	Jump to 3d branch
164	Jump to 4th branch

FIG. 3-8. Jump table for multiway branch.

For "one-shot" switch sensing, the computer senses the switch word periodically at a rapid rate. The computer is only to determine which switches have been *turned on*. The one-shot switch is not to be sensed when it is either left on or turned off. Thus one-shot switches are sensed according to Table 3-12. To code this, we can use the *extract* instruction. The old information is extracted with the new onto all units; then

TABLE 3-12. WHEN ONE-SHOT SWITCHES ARE SENSED

New information about switch	Old information about switch	Sensed
0	0	No
1	0	Yes
0	1	No
1	1	No

in the resulting word a zero will occur in the position corresponding to the switch that was just turned on. Hence, each time the switching word is sensed, it will be recorded in some other memory location. Then both the previous and the present switching word will be available at all times. To see why the extract instruction works, consider the following:

New word (α)	0	1	0	1
Old word (β)	0	0	1	1
(γ)	1	1	1	1
	1	0	1	1

Hence only when 0 in the old word corresponds to a 1 in the new word will a 0 result. Next, of course, the position of this zero must be determined. This can be accomplished by shifting left one position at a time, keeping track of the number of times shifted, until a zero is sensed in the most significant position.

Input-Output. Very often the input-output instructions of a computer are not adequate to read in a whole code or read out all of a given portion of the memory. In such a case a subroutine must be used for reading in and out of the computer. For example, consider the following read-in instruction for a one-address computer: "Read one word into α." If an entire code is to be read into successive addresses of the computer's memory, then clearly a subroutine that iterates this instruction is needed. For example, using such a subroutine, the following procedure might be followed in reading in a code, assuming that the subroutine is already in the computer's memory (see Fig. 3-9): The first instruction of the code to be read in will be a *jump* to the read-in subroutine. When this instruction is read into, say, address 000 and executed, then the computer will jump to the read-in subroutine. The read-in subroutine will read in the next two words of the code; these words will contain the initial and final addresses of the area in the computer's memory where the code is to be put. Then the subroutine loads the read-in instruction with the initial read-in address and the tally with the final read-in address, and the iteration of the read-in instruction begins. When the complete code has been read in, the subroutine will jump to the initial read-in address and com-

FIG. 3-9. Flow chart of input-output routine.

putation will proceed. It is probably wise to have the computer stop before computation proceeds, to give the computer operator a chance to turn off the input equipment, etc. Then, when the run button is pushed (see the next section), the computations will proceed.

Table Look-up. Consider the problem of coding a computer to look up a function table that has previously been stored in the computer's memory. That is, given a particular value of the independent variable, we desire a code that will select from the computer's memory the corresponding value of the dependent variable as recorded in the table.

Address	Operation	α	β
T_0		x_0	$f(x_0)$
T_1		x_1	$f(x_1)$
T_2		x_2	$f(x_2)$
T_3		x_3	$f(x_3)$
T_4		x_4	$f(x_4)$
T_5		x_5	$f(x_5)$
T_6		x_6	$f(x_6)$
T_7		x_7	$f(x_7)$

1, 2, 3

FIG. 3-10. Setup of table for table look-up routine.

METHOD 1. We have already illustrated above (see Sec. 2-7 under Instruction Modification) one simple technique that can be used when the independent variable in the table takes on successive integer values.

METHOD 2. Consider now the case of nonintegral values for the independent variable, but where the intervals between successive values of the independent variable given in the table are all equal. In this case, given the value of the independent variable for which the value of the function is desired, simply divide it by the interval to obtain an integer; thence this case reduces to the one considered in method 1.

The interesting case occurs when the values of the independent variable given in the table do not have equal intervals between them. Here, to be specific, suppose that we are in a two-address system and the table is recorded in successive cells of the computer's memory as follows: The value of the independent variable is found in the α position of a word, and the corresponding value of the dependent variable is found in the β position of the same word (see Fig. 3-10). Let us call the successive values of the independent variable $x_0, x_1, x_2, \ldots, x_f$, where of course $x_0 < x_1 < x_2 < \cdots < x_f$, and those of the dependent variable $f(x_0)$, $f(x_1), f(x_2), \ldots, f(x_f)$, and let us suppose that they are recorded in addresses $T_0, T_1, T_2, \ldots, T_f$. We desire to find $f(N)$ for some given value N, assuming that $N = x_i$ for some i; that is, no interpolation is considered here.

METHOD 3. One simple procedure is to compare successively $N:x_0$, $N:x_1$, $N:x_2$, . . . , until, for some i, $N = x_i$; then $f(N) = f(x)$ can be read from the computer's memory as desired. This, of course, can be accomplished by a simple iteration.

METHOD 4. There is another method that on the average will enable $f(N)$ to be obtained faster. The flow chart for this method appears in Fig. 3-11. The method consists in determining in which half of the table $x_i = N$ is located; then the halving procedure is again applied to this smaller part of the table, etc., until a single cell is left. For example, consider Fig. 3-10, and suppose that N is x_5. Then the first step of this

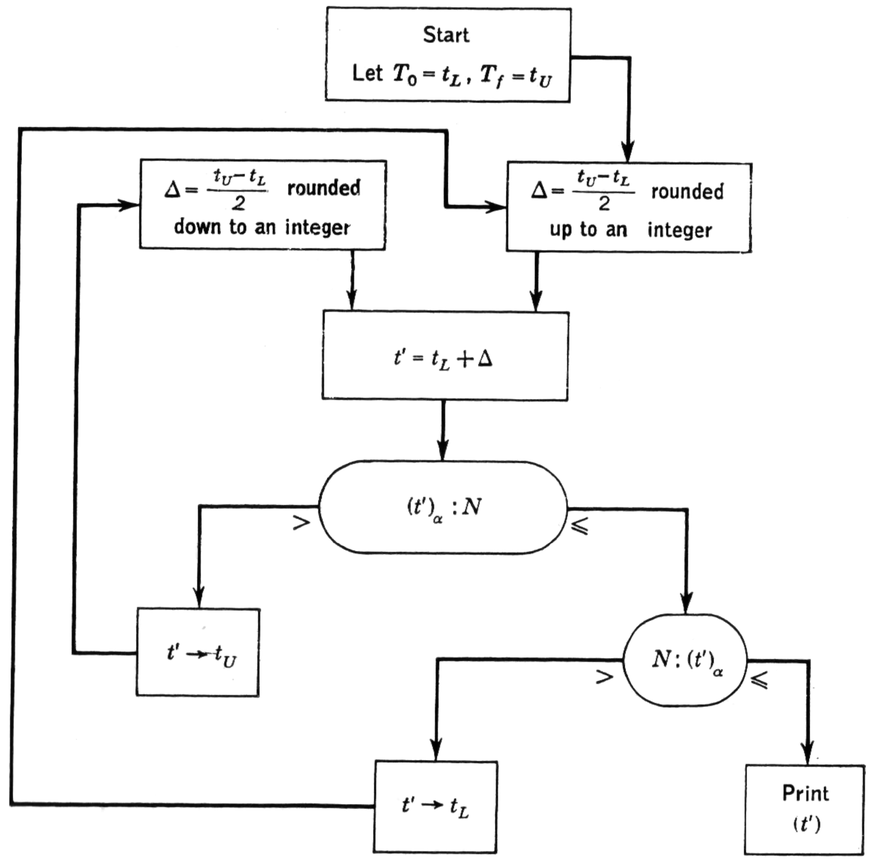


FIG. 3-11. Flow chart for table look-up, method 4.

method is to determine that N is included in T_4 through T_7. The second step locates N further, as being in T_4 or T_5. The third step in this case locates N as being in T_5. The code would be written as follows: At any step let t_L and t_U be the respective lower and upper addresses of the range of addresses in the table under consideration. We form $t' = t_L + \Delta$, the address that divides this range of address into two parts, where $\Delta = (t_U - t_L)/2$. Then we look to see whether N lies in the upper or lower half of the range; i.e., is the contents of the α part of t' greater or less than or equal to N, that is, $(t')_\alpha : N$? If $(t')_\alpha > N$, then t' becomes the upper address of the new range; that is, $t' \to t_U$. If $(t')_\alpha < N$, then t'

becomes the lower address of the new range; that is, $t' \rightarrow t_L$. Of course, if $(t')_\alpha = N$, we have the desired result. There is one little complication, however. When $\Delta = (t_U - t_L)/2$ is formed, it may not be an integer; but $t' = t_L + \Delta$ is an address and therefore must be an integer, hence must be rounded to an integer. A little reflection will show that, if $(t')_\alpha > N$, then the next Δ should be the nearest *smaller* integer to $(t_U - t_L)/2$, while if $(t')_\alpha < N$, the next Δ should be the nearest *larger* integer to $(t_U - t_L)/2$.

For example, consider Fig. 3-10 in connection with the flow chart of Fig. 3-11, and suppose for argument's sake that $N = x_5$ and that $T_0 = 100$, $T_1 = 101$, $T_2 = 102$, . . . , $T_7 = 107$. Then we would start with $t_L = 100$, $t_U = 107$, whence $\Delta = (107 - 100)/2 = 3\frac{1}{2}$, rounded up $= 4$. Hence $t' = 100 + 4 = 104$. Comparing $(104)_\alpha$:N, we find $N > (104)_\alpha$. Hence 104 is t_L, and we form $\Delta = (107 - 104)/2 = 1\frac{1}{2}$, rounded up $= 2$. Thus $t' = 104 + 2 = 106$. Comparing $(106)_\alpha$:N, find $N < (106)_\alpha$. Hence 106 becomes t_U. Thus $\Delta = (106 - 104)/2 = 1$, whence $t' = 104 + 1 = 105$. Comparing $(105)_\alpha$:N, find $N = (105)_\alpha$, and the result has been obtained.

3-7. The Control Panel

Purpose of the Control Panel. The control panel has three main functions: (1) to enable the operation and monitoring of the computer during the computation of a program; (2) to aid the checking, or "debugging," of a new program; and (3) to aid the maintenance of the computer by the engineer. The control panel and the program itself are the main communication links between the operator and the computer. By means of the control panel, the computer can be started and stopped, computations can be initiated, magnetic tapes can be properly positioned and other input-output equipment controlled, computer voltage and current levels, temperature, power input, etc., can be checked and adjusted, and so forth. There are wide variations in the features of control panels of different computers. In this section we shall discuss some of the possible features that a control panel might have (see Fig. 3-12).

In general, information is transmitted between the operator and computer by means of lights and buttons on the control panel. However, there is one method for monitoring a program during the computations using neither lights nor buttons that deserves special mention. It consists simply of a radio placed near the computer tuned to pick up signals generated during the computation. The computer, operating at a megacycle or more, presents a radio frequency; recursion loops generate audio-frequency modulations that can be detected. Various sound patterns will correspond to the different sequences or loops of instructions in the program. A coder who is computing many reruns of the same code, as

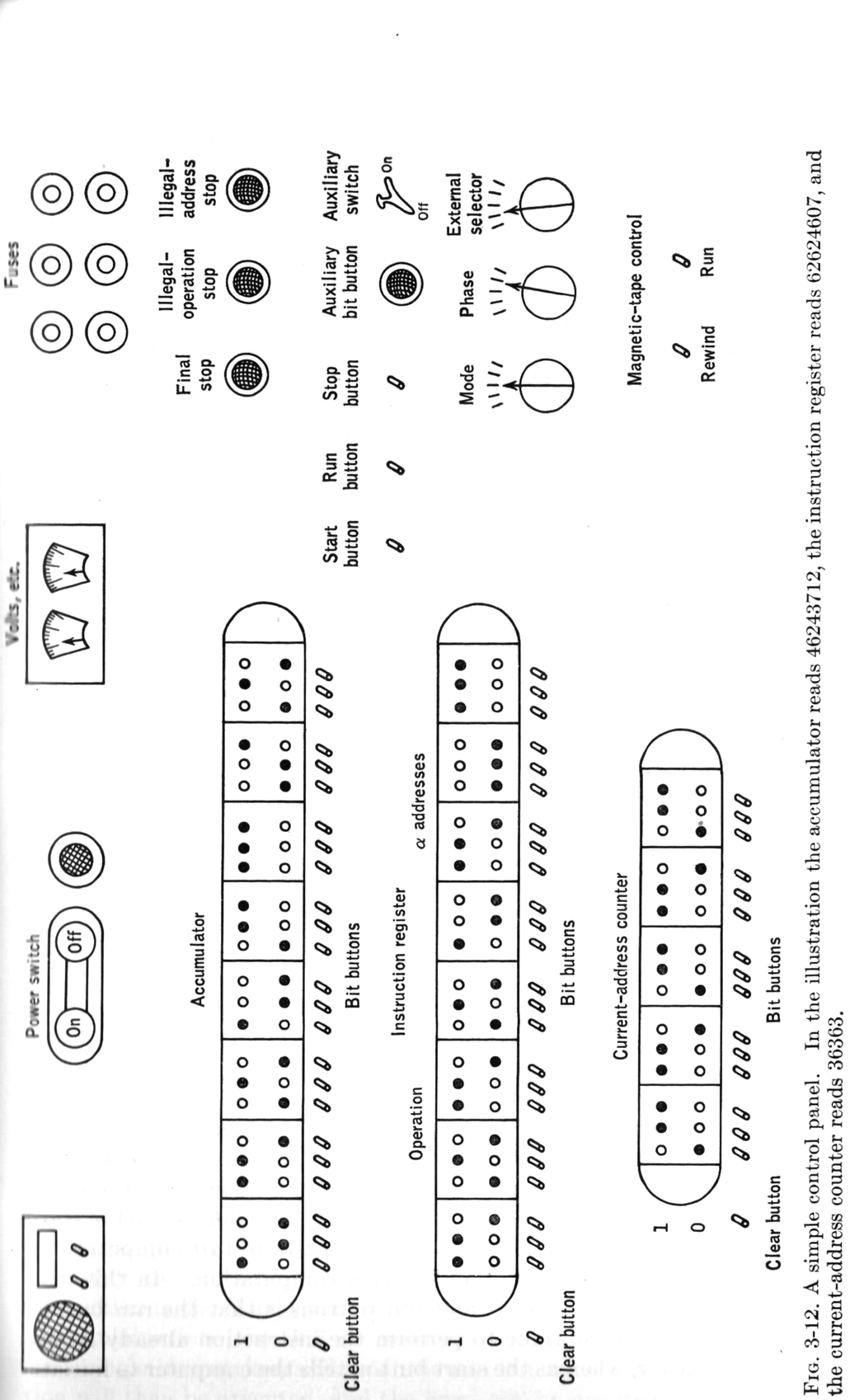


FIG. 3-12. A simple control panel. In the illustration the accumulator reads 46243712, the instruction register reads 62624607, and the current-address counter reads 36363.

would be the case in computing mathematical tables, can frequently relate these sounds to the various iterations of his program. A trained ear can then often detect errors or other troubles that his computations may have, just by "listening" to the program while it is being computed.

Lights and Signals. Usually one first notices the lights and signals displayed by the control panel. By means of these signals the computer can tell the operator something about its internal state. There are in general three kinds of lights and signals: the first kind tells why the computer stops during the course of a computation; the second presents a visual picture of the contents of various registers and memory cells; the third gives the electrical state of the computer.

When the computer stops or halts, usually one of several lights comes on, indicating the *cause for the stop;* i.e., a *stop* instruction has been executed (called a final stop), or an instruction being executed may contain an erroneous operation code not meaningful to the computer, or an instruction being executed may have reference to an erroneous memory address not legal for the computer,† or a check stop due to an auxiliary bit may have occurred, and so forth. Control panels are often constructed so that, when the computer has stopped or halted, the *contents of various registers* can be ascertained from sets of lights. For instance, there may be two small "grain-of-wheat" neon bulbs on the control panel corresponding to each bit of the accumulator. When the top bulb of the pair is on, the corresponding bit might be a unit; when the bottom bulb is on, the corresponding bit would be a zero. Similarly the contents of the instruction register of the current-address counter and of other registers and counters can be displayed. If an *electronic failure* has occurred, such as a blown fuse or a change in voltage or current readings beyond an allowable limit, appropriate signal lights will come on. Of course there will be a pilot light indicating that the power is on, etc.

Buttons and Switches. The only means the computer operator has to control the computer, other than by the code itself, is by the buttons and switches on the control panel. Besides the *on-off switch* for the power there is a *start* button, which initiates the computations. For example, when the start button is pushed, the computer may automatically take as the first instruction the contents of address 000. There is also a *stop* button; when it is pushed, computation ceases. In addition to the start button there may be a *run* button, which is used to restart computations after a halt has occurred in the middle of a computation. In this case the difference between the start and run buttons is that the run button tells the computer to continue to perform the instruction already in the instruction register, whereas the start button tells the computer to initiate

† Frequently the full complement of memory address is not constructed for a computer. Hence an erroneous instruction could refer to an address that the computer does not have.

a particular standard predetermined sequence of events, e.g., take 000 as the next instruction.

The computer can usually be operated in several different *modes.* For example, by setting a switch the computer can be made to execute only a single instruction each time the start or run button is pushed. In this way a sequence of instructions can be "stepped through" slowly by successively pushing the run button. This would be done, for example, if one desired to see several of the instructions and their results on the lights of the control panel that display the contents of various registers. On the other hand, a different setting of this mode switch can make the computer sequence itself through the instructions of a program at a very much reduced rate. Finally, of course, the mode switch can be set so that the computations proceed at full speed. Often in the single instruction mode, there is a *phase-selector* switch which makes the computer proceed through just *one phase* each time the run button is pushed.

So much for the special methods of starting the computer; consider next special methods for stopping or halting the computer. In previous sections we have observed how the operation of manual switches can be used to halt the computation when auxiliary bits are present. These switches are called *auxiliary switches.* There can be several auxiliary switches, each associated with a different auxiliary bit of the instruction word. Then, by setting the proper auxiliary switch, the computer can be made to stop at different sets of instructions. Other switches may make the computer halt at a particular phase when the stop button is pushed.

One of the most important uses of a control panel is to insert words manually into particular memory locations. Such an occasion may arise, for example, when it is necessary to correct only one or two words in a long program that is already in the computer's memory. There are several methods used to accomplish this. One method is to have one button associated with each bit of the registers that may be displayed by lights on the control panel. When the computer has been halted, the contents of a register may be changed by means of these buttons. For example, there may be one "clear" button for each register that, when pushed, makes the contents of the respective register all zeros. Then the bit buttons of this register are pushed to insert units in the desired bit locations. Under these conditions the procedure for inserting a word into a particular memory position would be as follows: Halt the computer; manually insert the desired word in the accumulator; similarly insert an instruction in the instruction register that will transfer the contents of the accumulator into the desired memory location. Set the mode switch for a single instruction, and push the run button. The inserted instruction will then be executed, and the word will be inserted into the desired memory location. Other methods might make use of the external input equipment to read a single word into the desired memory address.

There is usually a set of switches associated with the input-output equipment. Although, as we have seen, the selection of input-output equipment can be made by the program itself, frequently provision is made on the control panel for the manual selection of external equipment. Other switches present a means for manually winding, rewinding, and positioning magnetic-tape units. More complicated external equipment such as cathode-ray or television displays of course require still additional controls.

Let us now illustrate a possible procedure for initiating a computation by means of our control panel. First the input medium is properly set in the input device. Next the input device itself is turned on, and the external selector is set to choose this input equipment. An instruction is inserted into address 000 that will read in the whole program. The accumulator, instruction register, and current-address counter are all cleared (why?). The mode is set at high speed, and then the start button is pushed. The computer will then execute the instruction found in 000 first, initiating the read-in of the program.

3-8. Exercises

3-1. By rearranging the order of one of the comparisons together with the order of some of the instructions in Table 3-2 above, one instruction can be eliminated—i.e., the code will take only 10 instructions instead of 11. Can you find this comparison and rearrange the instructions properly? (HINT: In address 0005 the instruction can be changed so as to jump directly to make W the largest number. But then some other instructions must be reordered.)

3-2. Redefine in detail for a three-address system the *add*, *subtract*, *multiply*, and *divide* instructions given in Sec. 2-4.

3-3. Do Exercises 2-32 to 2-37, using the three-address instructions defined in this chapter. Carry out each of the five steps of coding in every problem.

3-4. Relate sets of instructions written for Exercise 3-3 to their respective boxes of the flow diagrams.

3-5. Do Exercises 2-32 to 2-37, using the two-address instructions defined in this chapter.

3-6. Do Exercises 2-32 to 2-37, using the one-address instructions defined in this chapter.

3-7. Define one-plus-one instructions corresponding to the one-address instructions given in Sec. 3-2; then redo the example of Table 3-8. Have a significant number of instructions been saved?

3-8. Write a code in three-address decimal-coded binary that will determine the largest of three numbers.

In the following exercises, utilize the three-, two-, and one-address systems given in the Appendix, unless otherwise specified:

3-9. Consider the example illustrated by Fig. 2-8. Code this in a three-address system.

3-10. What would a counter-setting instruction be in a one-address system?

3-11. Code the same problem as in Exercise 3-9, using a one-address system.

3-12. Find the unnormalized and normalized sum of 77770000 $\times$ 2^5 and 00007777 $\times$ 2^6 (where, of course, the numbers are in octal).

3-13. Find the unnormalized difference of 0000 7777 $\times$ 2^6 and 7777 0000 $\times$ 2^5.

3-14. For the numbers 7777 0000 $\times$ 2^5 and 0000 7777 $\times$ 2^6 find:
The unnormalized major product, unrounded.
The unnormalized major product, rounded.
The normalized major product, unrounded.
The normalized major product, rounded.
The unnormalized minor product.
The normalized minor product.

3-15. What combination of other logical instructions would produce the same result as an *extract* instruction, in a binary computer?

3-16. How can *return-jump* instructions facilitate the use of subroutines?

3-17. Use the *modify-index-register-and-tally* instruction to code the problem of Table 2-15. Write the code in a two-address system.

3-18. In precisely what sense are indirect addressing and augmented addressing the same?

3-19. Explain why for addressing "gain" we must have $i \geqq r$, $R \geqq m$ for indirect addressing and $r + i < R + i$, $R + i \geqq m$ for augmented addressing, where there are 2^r high-speed registers of contents R bits, 2^m memory words, and i bits in the instruction address.

3-20. Code both the split-tree and jump-table method for a four-way branch, using a one-address instruction system.

3-21. Suppose that only a single on-off switch could be on at one time. Code a subroutine that will determine the position of a switch that is turned on in a one-address system.

3-22. Write a code for the same problem as Exercise 3-21, but now consider one-shot switches.

3-23. Write an input subroutine, using the one-address input instruction given in this chapter.

3-24. Using a one-address system, with an input instruction that reads in one word at a time as described in this chapter, write a read-in subroutine.

3-25. Write a code for method 3 of the table look-up problem. Make use of an *extract* instruction to extract the successive x_i onto all zeros to compare with N; then extract $f(N)$ onto zeros for the read-out.

3-26. Code method 4 of the table look-up problem in a two-address system.

3-27. Suppose that a characteristic "click" can be heard on the radio monitor each time the computer goes through a loop. Suppose also (for simplicity in this exercise) that any instruction takes precisely 100 μsec to execute on a one-address computer. Write a code that will play the first few notes of the song "Dixie." (HINT: Look up the audio frequencies of the first few notes of "Dixie," and code successive loops that will repeat at these frequencies.)

3-28. Describe a procedure for finding out, by means of the control panel, what word is in a particular memory address.

3-29. Suppose that a subroutine were necessary in order to read in a program. What would be a procedure for initiating a program by means of the control panel?

3-30. If a computer needs a read-in subroutine, how is the read-in subroutine itself read in (for the very first time)? (HINT: See, for example, chap. 9 in R. S. Ledley, "Digital Computer and Control Engineering," McGraw-Hill Book Company, Inc., New York, 1960.)

3-31. Write a code of one-address instructions for use in iterative processes, utilizing the flow chart of Fig. 9-6, assuming that $f(X_n)$ is a subroutine.

3-32. Write a code for Aitken's method of Sec. 9-4, using two-address instructions. Assume that the code is to become a subroutine in a library of subroutines.

3-33. Write codes using a one-address instruction system for computing the derivatives of $y = f(x)$, using the formulas of Sec. 9-7.

3-34. Write a subroutine using a three-address instruction system for solving the differential equation

$$\frac{dy}{dx} = f(x,y)$$

using the first method given in Sec. 9-9.

3-35. Write a code for solving Laplace's equation, using a one-address instruction system and the method of iteration of difference equations of Sec. 9-10.

3-36. Draw a flow chart for evaluating Laplace's equation by the relaxation method given in Sec. 9-10.

3-37. Write a three-address code for determining the prime implicants of a Boolean function, given the designation number of the function (see Sec. 10-7).

3-38. Write a one-address code for determining the prime implicants directly by the method of Sec. 10-8.

3-39. Write a one-address code for determining whether or not two Boolean functions are independent (see Sec. 10-10).

3-40. Write a subroutine, using one-address instructions for Boolean matrix multiplication.

3-41. Write a subroutine, using one-address-format instructions that will solve Boolean equations. Assume that the designation numbers of each side of the equations are given and that just the designation numbers of the solutions are to be generated.

3-42. Write a one-address program for the flow chart of Fig. 11-9.

3-43. Write a three-address code to correct three errors, using the scheme illustrated in Fig. 11-15.

3-44. Code the formulas given in Sec. 11-7 for evaluating the probability of false drops in irredundant codifying, using a one-address instruction system.

3-45. *The Minimum Computer.* It is clear that a general-purpose computer does not really "need" the multiplication or division instructions, for these operations can be formed by a combination of additions or subtractions. Similarly other operations can be omitted and replaced by simpler ones. The question arises: What are the minimum instructions necessary for a general-purpose computer—or, indeed, what criteria can be applied to determine whether or not a computer is actually a general-purpose computer? These questions have long been of interest to mathematicians and logicians. The so-called Turing machine, or "universal computer," was developed for this purpose (see A. M. Turing, On Computable Numbers, with an Application to the Entscheidungs Problem, *Proc. London Math. Soc.*, ser. 2, vol. 42, pp. 230–265; a more readable description is given in Martin Davis's "Computability and Unsolvability," chap. 1, McGraw-Hill Book Company, Inc., 1958). More recently S. P. Frankel has considered the minimum general-purpose computer; see his On the Minimum Logical Complexity Required for a General Purpose Computer, *IRE Trans. on Electronic Computers*, vol. EC-7, no. 4, pp. 282–285, December, 1958. Frankel's computer (called the *Microcephal AC*) can only subtract. Can you program multiplication on Frankel's minimum computer?

REFERENCES

Astrahan, M. M.: The Role of Large Memories in Scientific Communications, *IBM J. Research Develop.*, vol. 2, no. 4, pp. 310–314, October, 1958.

Blaauw, G. A.: Indexing and Control-word Techniques, *IBM J. Research Develop.*, vol. 3, no. 3, pp. 288–301, July, 1959.

Brooks, F. P., Jr.: A Program-controlled Program Interruption System, *Proc. Eastern Joint Computer Conf.*, December, 1957, pp. 128–132.

———, G. A. Blaauw, and W. Buchholz: Processing Data in Bits and Pieces, *IRE Trans. on Electronic Computers*, vol. EC-8, pp. 118–124, June, 1959.

Buchholz, W.: The Selection of an Instruction Language, *Proc. Western Joint Computer Conf.*, May, 1958, p. 128.

Cocke, J., and H. G. Kalsky: The Virtual Memory in the STRETCH Computer, *Proc. Eastern Joint Computer Conf.*, 1959, pp. 82–93.

Douglas, A. S.: Techniques for the Recording of, and Reference to Data in a Computer, *Computer J.*, vol. 2, pp. 1–9, April, 1959.

Eckert, J. P., J. C. Chu, A. B. Tonik, and W. F. Schmitt: Design of Univac-LARC System: I, *Proc. Eastern Joint Computer Conf.*, 1959, pp. 59–65.

Fredkin, Edward: Trie Memory, *Communs. ACM*, no. 9, pp. 490–499, September, 1960.

Holland, J. H.: On Iterative Circuit Computers Constructed on Microelectronic Components and System, presented at the Western Joint Computer Conference, May, 1960.

Isaac, E. J., and R. C. Singleton: Sorting by Address Calculation, *J. Assoc. Computing Mach.*, vol. 3, pp. 169–174, 1956.

Kilburn, T.: The University of Manchester High-speed Digital Computing Machine, *Nature*, no. 164, p. 684, 1949.

Ledley, R. S.: Organization of Large Memory Systems, in M. Yovits (ed.), *Proc. Symposium on Large Capacity Memory Techniques for Computing Systems*, The Macmillan Company, New York, 1962.

McDermio, W. L., and H. E. Peterson: A Magnetic Associative Memory System, *IBM J. Research Develop.*, vol. 5, no. 1, pp. 59–63, January, 1961.

Newell, Allen: On Programming a Highly Parallel Machine to Be an Intelligent Technician, presented at the Western Joint Computer Conference, May, 1960.

Rajchman, J. A.: A Myriabit Magnetic Core Matrix Memory, *Proc. IRE*, vol. 41, pp. 1457–1471, October, 1953.

———: Computer Memories—A Survey of the State-of-the-art, *Proc. IRE*, vol. 49, no. 1, pp. 104–127, January, 1961.

Shaw, J. C., A. Newell, H. A. Simon, and T. O. Ellis: A Command Structure for Complex Information Processing, *Proc. Western Joint Computer Conf.*, May, 1958, p. 119.

Swanson, J. A.: Physical versus Logical Coupling in Memory Systems, *IBM J. Research Develop.*, vol. 4, no. 3, pp. 305–311, July, 1960.

Computer Manuals

Honeywell 800 Programmers' Reference Manual.

IBM 705 Electronic Data-processing Machines, 1956, pp. 5, 6, 16, 17, 109.

IBM 650 Magnetic Drum Data-processing Machine, Manual of Operation, 1955, pp. 10-13.

IBM Reference Manual 1401, 1960, p. 12.

IBM Reference Manual 7090, Data Processing, 1959, pp. 8, 18–67.

UNIVAC Model 1103 A, Scientific Programming Manual, 1958, pp. 2–7.

UNIVAC-LARC (general description), 1959, pp. 11–14.

CHAPTER 4

PROGRAMMING FOR SPECIAL-PURPOSE DIGITAL COMPUTERS

4-1. Introduction

To increase the effectiveness of digital computers in performing certain special functions, it is often advisable to introduce special-purpose features into the design of the computer. These special-purpose features will of course influence the programming methods required in the use of the computer. In this chapter we shall consider some of the problems involved in programming for such special-purpose computers. The method of presentation is to consider specially selected examples of special-purpose computers, ranging from those which differ radically from the general-purpose computer considered above to those built upon the extended general-purpose concepts. It is our hope that examples of such computers—some having no formal instruction systems at all, some having no arithmetic functions, some having great deviations from the conventional instruction format, and some having multiple code-handling capabilities—will present a broad view of the many possibilities existing in special-purpose computer programming. The examples discussed are not necessarily based on actually existing special-purpose computers but have been formulated to illustrate a large variety of important concepts in special-purpose computer programming.

Outline. We shall consider first the digital differential analyzer. This computer has no instruction system but rather consists of a collection of computer units that are wired together as required by each particular problem. Next we shall consider a logical-control computer, using the control of a chemical factory as an example. Here a special "external-control" instruction is introduced to facilitate the logical computations. A special design for a searching computer will then be considered. This special-purpose computer can be incorporated in a general-purpose computer as well. Business and logistics problems illustrate the need for special features enabling the handling of large arrays of numbers. Here a radically different instruction format is illustrated. Finally we consider the multiple-general-purpose computer, which is essentially an interlocked collection of computers, used in such a manner as will take

greatest advantage of the special-purpose components of which it is comprised. In our illustration this multiple-purpose computer will be comprised of the computer components discussed in the previous sections.

4-2. The Digital Differential Analyzer

Purpose. Frequently it is necessary to compute the value of some function point by point. For example, in the automatic-milling-machine application described in Chap. 1, the computer attached to the milling machine must guide the tool from point to point along a predetermined path that was given only grossly at the input, e.g., a linear interpolation between two points or a circle of a certain radius. Similarly in many other control operations rapid calculations of $\sin \theta$ and $\cos \theta$ may be necessary, as in accurately sweeping an area with an antenna or in computing some functions. Not infrequently the familiar analog techniques cannot produce a sufficient number of significant figures for the accuracy required. In such cases one can use a digital differential analyzer as a small, relatively simple and inexpensive special-purpose digital computer.

The Concept of the Digital Differential Analyzer. The digital differential analyzer computes successive values of a function by means of successive *differential additions.* Consider the problem of computing a table of values for a function $y = f(x)$. If we start with a given $y(x_0)$, then $y(x_1) = y(x_0) + [y(x_1) - y(x_0)] = y(x_0) + (\Delta y)_{x_0}$. Similarly we write $y(x_2) = y(x_1) + (\Delta y)_{x_1}$, and so forth. In other words,

$$y(x_{i+1}) = y(x_i) + (\Delta y)_{x_i}$$

Now suppose that $y = e^x$. Then $dy = e^x\,dx = y\,dx$; or approximately

$$(\Delta y)_{x_i} = y(x_i)(x_{i+1} - x_i) = y(x_i)\,\Delta x$$

Thus we can compute successive approximate values of e^x by

$$y(x_{i+1}) = y(x_i) + y(x_i)\,\Delta x$$

The smaller we make Δx, the more accurate our results will be. However, there is an obstacle here. Remember that multiplication produces a *double-length result,* so that, if Δx is small enough, the major part of $y(x_i)\,\Delta x$ will be zero. Thus errors will result when the minor product, which contains the significant figures of $y(x_i)\,\Delta x$, is dropped. It would therefore be better to work with the double-length extension of $y(x_i)$. However, this is not particularly desirable for what is supposed to be a *small* special-purpose computer.

The use of a double-length accumulator can be avoided; to find out how, let us examine our procedure more carefully. As we have noted, each time we form $y(x_i)\,\Delta x$ the major product is zero; but certainly the major part of $y(x_i)$ must change eventually. This must occur during

the addition $y(x_i) + y(x_i)\,\Delta x$ and will be the result of a *carry* from the minor part into the major part. In other words, as we accumulate in the minor part, we eventually propagate a carry, or overflow, from the minor part of $y(x_i)$ into the major part. If we are working in *binary*, the carry can only be a 1 and the major part of $y(x_i)$ can change at most by 1 during any iteration. Hence we do not need a double-length accumulator at all—we need a *single-length accumulator* that simply accumulates successive minor parts of $y(x_i)\,\Delta x_i$ and a *counter* that holds the major part of $y(x_i)$ and *adds* 1 to the major part of $y(x_i)$ each time there is an *overflow* from the minor accumulator. We call the accumulated minor parts the *residual.*

True multiplication is not essential in finding $y(x_i)\,\Delta x$, for we can always take $\Delta x = 1/2^q$, and we need only shift $y(x_i)$ by q positions to the right to form $y(x_i)\,\Delta x$. But since we do not have a major part to our accumulator, we can always choose Δx so that the shifting of $y(x_i)$ need not have to occur *actually* (although of course it occurs *virtually*).

To illustrate these points, consider a 4-bit binary word and let $\Delta x = .0001$ (see Fig. 4-1). In Fig. 4-1*a* we have illustrated the accumulator that is to hold the residual and the counter that is to hold the major part of the function value y. The circle represents the component wherein y is *virtually* but *not actually* shifted; this is simply a gate that at the proper time passes y to be accumulated with the previous residual to form the new residual. The dash-dot lines represent the "true" juxtaposition of the double-length value of y. For our illustration, start with $x_0 = 0$ and $y_0 = e^0 = 1$ preloaded into the counter. Then $y(x_0) = 01.00$ and $y(x_0)\,\Delta x = 00.00\ 0100$, where 0100 is the residual. Since

$$y(x_1) = y(x_0) + y(x_0)\,\Delta x = 01.00\ 0100$$

the 01.00 remains unchanged in the counter and 0100 is put into the accumulator (see Fig. 4-1*b*). With $y(x_1)\,\Delta x = 00.00\ 0100$ again, we have $0100 + 0100 = 1000$ as the residual, and 01.00 remaining still unchanged in the counter as $y(x_2)$. Then $y(x_3)\,\Delta x = 00.00\ 0100$, whence $1000 + 0100 = 1100$ is the new residual and 01.00 remains unchanged as $y(x_4)$ in the counter. Next $y(x_4)\,\Delta x = 00.00\ 0100$, whence $1100 + 0100 = \textit{carry } 1 + 0000$; now the counter is increased by 1, putting 01.01 in the counter as $y(x_5)$ and leaving 0000 as the new residual. We continue with $y(x_5)\,\Delta x = 00.00\ 0101$, and so forth. The result of each step and the graph of the function so calculated are shown in Fig. 4-1*b*.

Computing Units. The combination of accumulator, gate, and counter is called a computing unit of a digital differential analyzer. Many units can be connected together to compute various functions. The contents of the accumulator, the residual, is denoted by R; the contents of the

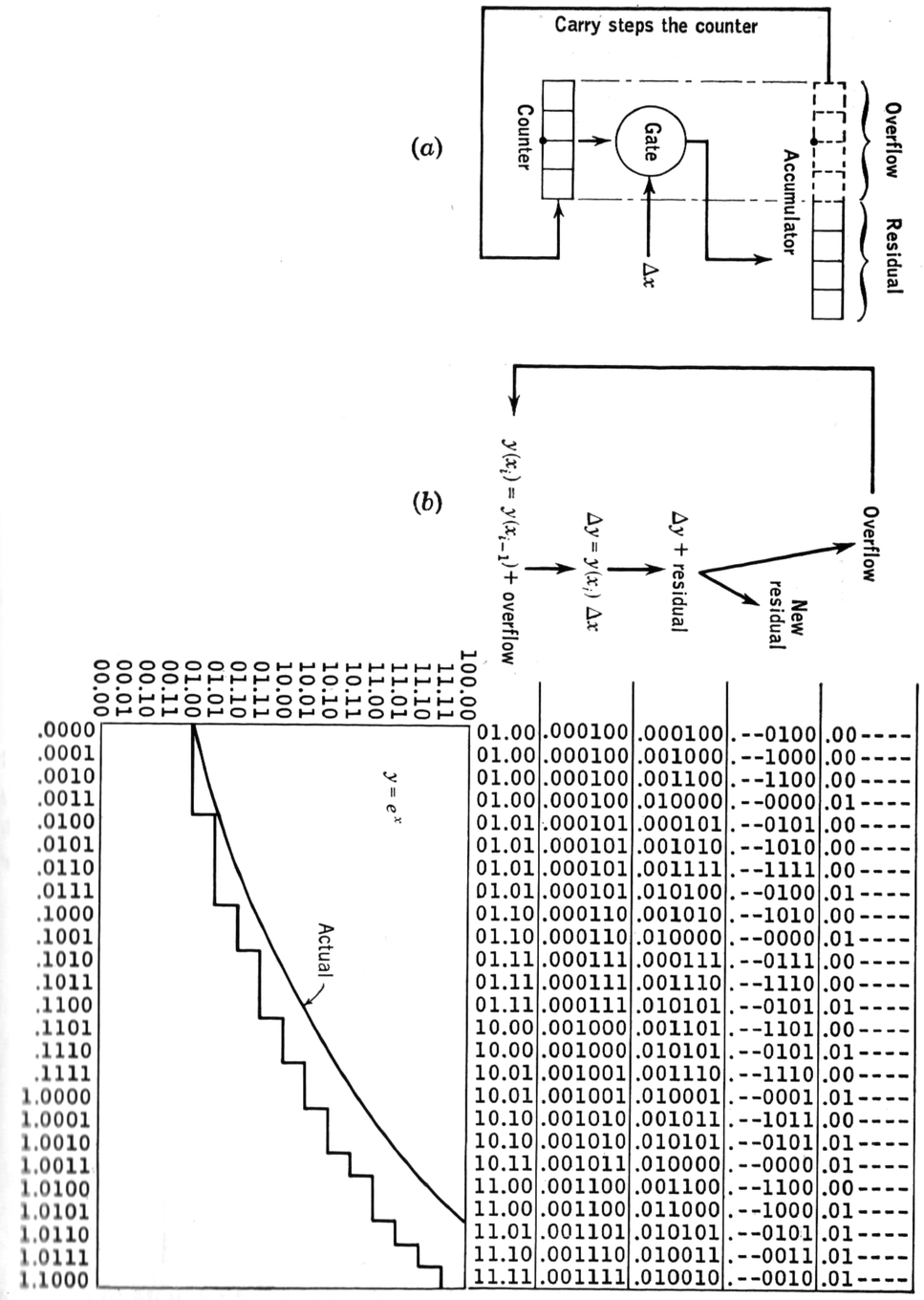


FIG. 4-1. Concept of the digital differential analyzer.

counter, the functional value, is denoted by y; the overflow, or carry, from the accumulator that steps the counter, i.e., the differential value that is to be added to some functional value, is denoted by Δy (see Fig. 4-2). The gate may cause either addition or subtraction of the contents of the counter (virtually but not actually shifted) to the accumulator.

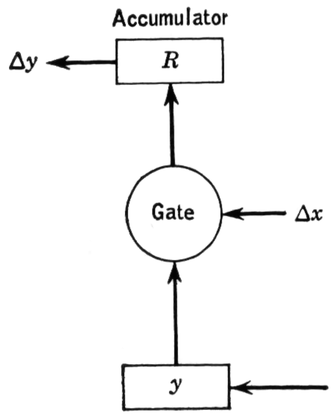


FIG. 4-2. Computing unit of digital differential analyzer.

As an example of how units can be combined, consider now the problem of computing sin x and cos x. Let

$$y_1 = \cos x \qquad \text{and} \qquad y_2 = \sin x$$

Then the differential equations are

$$dy_1 = -\sin x\, dx = -y_2\, dx$$
$$\text{and} \qquad dy_2 = \cos x\, dx = y_1\, dx$$

Thus we have

$$y_1(x_{i+1}) = y_1(x_i) + \Delta y_1 \qquad \Delta y_1 = -y_2\, \Delta x$$

and

$$y_2(x_{i+1}) = y_2(x_i) + \Delta y_2 \qquad \Delta y_2 = y_1\, \Delta x$$

Hence the arrangement shown in Fig. 4-3 will compute y_1 and y_2.

Programming. There are three steps to programming a problem: First, the proper differential equations must be worked out so that Δy_j appears in the form $y_i\, \Delta x$. Second, the connections between units must be diagramed. And finally the scaling must be worked out. Consider

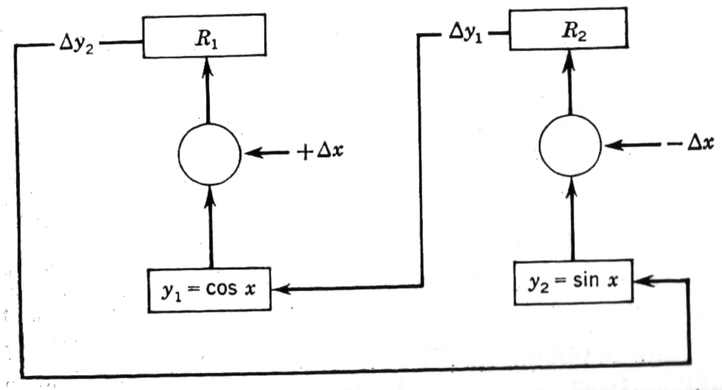


FIG. 4-3. Combinations of digital-differential-analyzer computing units to compute $y_1 = \cos x$ and $y_2 = \sin x$.

this latter step first. Note that, *since the counter can be increased only by 1 at a time*, then during any iteration y can increase only by 1. In other words, the graph of the function *cannot be steeper* than 45°. Thus in our example of Fig. 4-1 we could not compute e^x for any greater values of x.

However, the slope of any function can be adjusted by scaling so that it does not exceed 45° within the range of computation.

Consider as another example the problem of computing the product $y_3 = y_1 y_2$. Since $dy_3 = y_1\, dy_2 + y_2\, dy_1$, we have

$$y_3(x_{i+1}) = y_3(x_i) + \Delta y_3 \qquad \Delta y_3 = y_1\, \Delta y_2 + y_2\, \Delta y_1$$
$$y_2(x_{i+1}) = y_2(x_i) + \Delta y_2$$
$$\text{and} \qquad y_1(x_{i+1}) = y_1(x_i) + \Delta y_1$$

Hence the configuration of Fig. 4-4 will compute $y_3 = y_1 y_2$. Note that

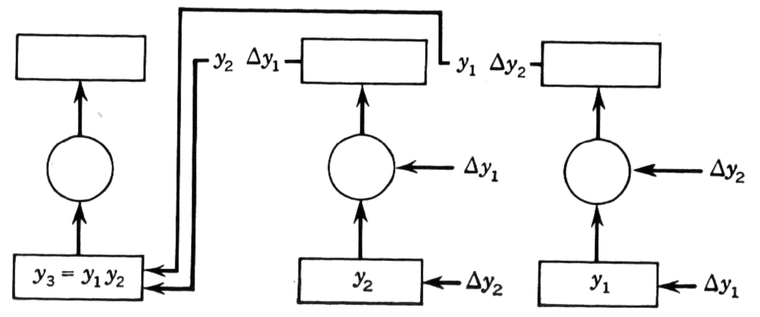


FIG. 4-4. Computing $y_3 = y_1 y_2$.

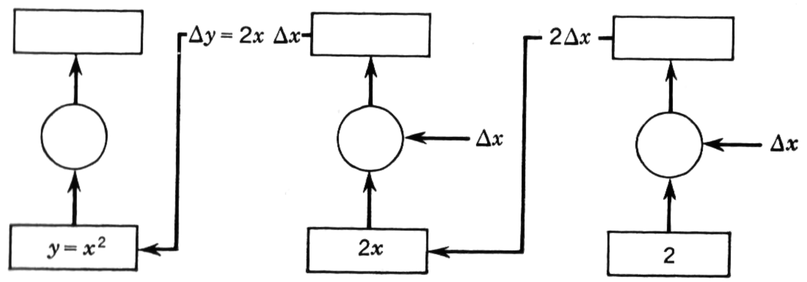


FIG. 4-5. Computing $y = x^2$.

the counter for y_3 has two inputs and that these must be electronically arranged so that they do not step the counter at precisely the same time. Note also that not all of the computing unit for y_3 is used. Another feature of this arrangement is that the inputs to the gates are not a single Δx, but rather Δy_1 and Δy_2 themselves.

Next consider the computation of $y = x^2$. Here $dy = 2x\, dx$ and $y(x_{i+1}) = y(x_i) + \Delta y$, $\Delta y = 2x_i\, \Delta x$; Fig. 4-5 shows the arrangement. Here note that one of the counters is loaded with the constant 2 and never changes.

Finally consider the computation of $y = 1/x$, that is, division. Here

$dy = (-1/x^2)\,dx$; letting $w = -1/x^2$, we find $dy = w\,dx$ and

$$dw = \frac{2}{x^3}\,dx = -\frac{2}{x}\left(-\frac{1}{x^2}\,dx\right) = -2y\,dy$$

Thus $\quad y(x_{i+1}) = y(x_i) + \Delta y \qquad \Delta y = w\,\Delta x$

and $\quad w(x_{i+1}) = w(x_i) + \Delta w \qquad \Delta w = -2y\,\Delta y$

The arrangement of units for this computation is left as an exercise.

4-3. Real-time Logical Systems Control: A Real-time Control Computer

An Example. Very often the primary function of an on-line digital control system (i.e., a system connected to a dynamic process) is to make sequences of logical decisions that depend on the state of certain variables associated with the world external to the computer. Here we shall consider the special-purpose real-time logical-control computer and show how it might be applied in a specific example of controlling a chemical factory. In discussing first the specific application, the reasons for the various aspects of the control computer will be better appreciated. Although the example is concerned with the control of a chemical factory performing a specific chemical process, the situation is conceptually similar in almost all such real-time control problems as may occur in other types of automatic factory control, in tactical gunnery control, or in distributing tracking data to command centers according to predetermined strategies. The example also contains a feedback loop by which the results of the input digital disposition influence the outside world, which in turn again influences the input to the computer.

Figure 1-9 on page 16 shows the flow chart of a chemical factory manufacturing Al_2O_3. Briefly, clay is mixed with water and a detergent to make a wash containing the raw material. A certain amount of this wash is tapped off into an evaporation tank, where it is concentrated by evaporation. Then in another tank HCl gas is mixed with the wash until it is saturated. The wash is centrifuged and a precipitate of $AlCl_3{\cdot}6H_2O$ is obtained. The precipitate is decomposed, by heating, into gaseous HCl, which is reused, and into solid Al_2O_3, the desired end product. It is of course understood that, even though this problem is based on an actual process, it has here been grossly oversimplified and actually stylized. It is not meant to be examined from a chemical-engineering point of view, since many details of both the necessary conditions and desired results are but casually indicated.

The state of the chemical factory is given by 16 signals, listed in Table 4-1. The computer periodically senses these signals, generating from them 12 control signals that direct the future state of the factory. The future state of the factory results in changes of the input signals, whence new control signals are generated by the computer, and so forth. Table

4-2 gives the relation between the sensed-signal conditions and the necessary control signals to be generated. A symbolic notational form for each statement is also given, where · means *and* and ¯ means *not* (i.e., no signal).

Programming Logical Control. The computer will have one subroutine for each set of the control signals to be generated. Hence a jump table is

TABLE 4-1. SIGNALS SHOWING THE STATE OF THE CHEMICAL FACTORY

Signals generated by factory	*Symbolism*
Level of $L(1)$ reached	**L(1)**
Concentration test good	**C**
Time for concentration test	**T(C)**
Valve 4 open	**V(4)**
Level of $L(2)$ reached	**L(2)**
Tank 2 empty	**E(2)**
Tank 3 full	**F(3)**
Temperature $T(1)$	**T(1)**
Temperature $T(2)$	**T(2)**
Saturation test positive	**S**
Tank 3 empty	**E(3)**
Valve 7 open	**V(7)**
Centrifuging finished	**C(F)**
Centrifuge empty	**E(C)**
Pressure of HCl too high	**P**
Enough volume	**W**

required to initiate the proper subroutine for a given state of the sensed signals. However, we need an input to the jump table that tells which *jump* instruction to use, and we shall first describe how this input may be accomplished.

At any instant the bits in the signal-sensing word (Fig. 1-9) tell which signals are on and which are off. For instance, the word might be as in Table 4-3; in the table under each bit of the word is shown the state of the corresponding signal. We can similarly use a word for each of the significant sensed-signal conditions of Table 4-2, so that the conditions can be matched with the state of the sensed signals at any time. For example, the first condition, **L(1)·C̄·T(C)** · · · , would be represented by 101 · · · . But what about the rest of the bits? Condition **L(1)·C̄·T(C)** does not mention the rest of the signals because for that condition they make no difference. They are to be ignored. Thus we need another word that tells what positions are to be considered and what positions are to be ignored. The second word will have units corresponding to bits that are to be considered and zeros corresponding to bits to be ignored. For example, **L(1)·C̄·T(C)** is denoted by the following two words:

1st word:	101*x*	*xxxx*	*xxxx*	*xxxx*
2d word:	1110	0000	0000	0000

where the x means that it makes no difference whether the bit is zero or unit. Similarly the fourth condition, $\bar{\mathbf{V}}(\mathbf{4})\cdot\mathbf{L}(\mathbf{2})\cdot\mathbf{E}(\mathbf{3})$, is denoted by

1st word:	$xxx0$	$1xxx$	$xx1x$	$xxxx$
2d word:	0001	1000	0010	0000

There would then be two words representing each of the 10 conditions of Table 4-2.

TABLE 4-2. CONTROL SIGNALS

Sensed-signal conditions	Symbolic representation	Corresponding signals to be generated
1. Level is at $L(1)$, *and* concentration test is bad, *and* time for testing is due	$L(1)\cdot\bar{C}\cdot T(C)$	Throw away solution by opening valve 3, and add clay, detergent, and water in tank 1 by opening hatch 1, valves 1 and 2
2. Level is at $L(1)$, *and* concentration test is good, *and* time for testing is due, *and* tank 2 is empty	$L(1)\cdot C\cdot T(C)\cdot E(2)$	Tap off certain amount of fluid into tank 2 by means of valve 4
3. Level $L(1)$ has not been reached, *and* valve 4 is not open	$\bar{L}(1)\cdot\bar{V}(4)$	Add detergent, water to tank 1 by opening valves 1 and 2
4. Valve 4 is not open, *and* level is $L(2)$, *and* tank 3 is empty	$\bar{V}(4)\cdot L(2)\cdot E(3)$	Transfer fluid to tank 3 by opening valve 5
5. Temperature is $T(1)$, *and* tank 3 is full, and valve 7 is off	$T(1)\cdot F(3)\cdot\bar{V}(7)$	Cut rate of HCl flow to slow by closing valve 6
6. Temperature is $T(2)$, *and* tank 3 is full, *and* valve 7 is off	$T(2)\cdot F(3)\cdot\bar{V}(7)$	Increase rate of HCl flow to fast by opening valve 6
7. Saturation test is good, *and* valve 7 is closed, *and* centrifuge is empty	$S\cdot\bar{V}(7)\cdot E(C)$	Close flow of HCl by means of valve 6, transfer fluid to centrifuge by valve 7, and start centrifuge
8. Centrifuge finished.....	$C(F)$	Put precipitate in furnace by opening hatch 2, throw away wash by opening valve 8
9. Pressure of HCl high...	P	Lower heat of furnace
10. Weight is enough.....	W	Take out resultant Al_2O_3 by opening hatch 3

TABLE 4-3. EXAMPLE OF SIGNAL-SENSING WORD

1	0	1	0	0	0	1	1	0	0	0	0	1	0	1	0
$L(1)$	$\bar{C}$	(TC)	$\bar{V}(4)$	$\bar{L}(2)$	$\bar{E}(2)$	$F(3)$	$T(1)$	$\bar{T}(2)$	$\bar{S}$	$\bar{E}(3)$	$\bar{V}(7)$	$C(F)$	$\bar{E}(C)$	P	$\bar{W}$

Figure 4-6 is a flow chart of a real-time operating program that will compare each of the conditions with the sensed-signal word until a favorable comparison is obtained; then, depending on which condition compared favorably, a jump will be made to the proper signal-generating

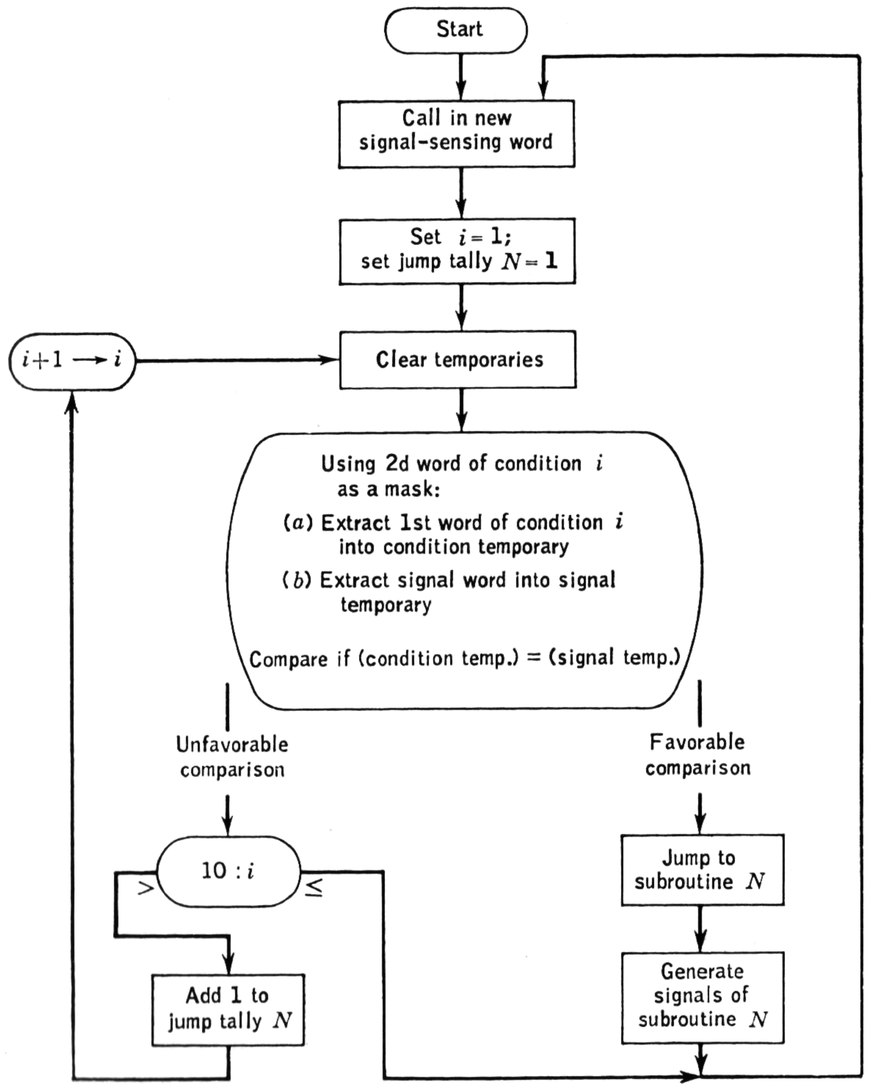


FIG. 4-6. Flow chart of real-time operating program.

subroutine. After completing the subroutine, a new sensed-signal word will be called into the high-speed memory, and the process will start over. If no condition compared favorably, the program would call in a new sensed-signal word and repeat the comparisons, and so forth.

Description of a Special-purpose Logical-control Computer. When hundreds of signals must be sensed, as would occur in a more realistic

case, it may be more efficient to have the two extractions and comparison made in one step by a special computer instruction, which we shall call an *external-control instruction*. Such an instruction would have three inputs: the sensed-signal word and the first and second condition words. If the comparison is unfavorable, the instruction would automatically increase a relative counter. This instruction would repeat itself as the next instruction. The addresses of the two condition words are taken relative to this counter so that they are automatically modified appropriately for the next condition when the external-control instruction repeats

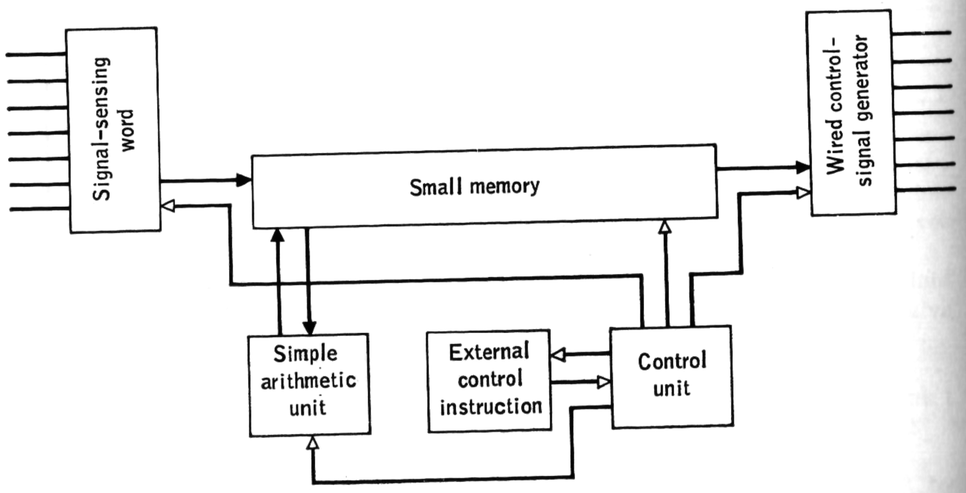


FIG. 4-7. Real-time logical-control computer.

itself. If the comparison is favorable, the next instruction is taken from the address of the external-control instruction plus the counter reading; i.e., the external-control instruction would be followed by the jump table. The method by which the comparisons are repeated is left as an exercise for the student.

For a permanent installation the subroutines generating the control signals can be wired in. Hence the real-time logical-control computer need contain little more than the external-control instruction, an instruction to clear the relative counter, a *comparison* instruction, a *jump* instruction, and an *add* instruction (see Fig. 4-7).

Since the program will not often be changed, it can be inserted into the computer by means of the control panel; therefore no input-output equipment is necessary except the signal-sensing word and the wired control-signal generator. The memory of such a special-purpose computer need be only a little larger than necessary to accommodate the condition words. The speed of the memory need only be fast compared with the feedback reaction time of the system it is controlling. For the

chemical factory here described this need not be particularly fast; for a tracking system it must be very high speed.

4-4. Special-purpose Information-retrieval Computer

An Example. The occasion may frequently arise where a special-purpose searching, sorting, and ordering computer becomes feasible. As a specific example of the use of such a computer, which will indicate the motivation for its design, consider a cost-accounting system in a factory. Each workman in the factory is given a job; he procures supplies and parts; he spends time using the supplies and assembling the parts to produce some final object; then he may be given another job (which may be a repetition of the first), etc. The executives and accountants of the factory would like to know how many objects of each type have been completed, how many supplies and how many parts were used during a particular period of time, how many hours a particular workman spent on a particular job, how many man-hours altogether were spent in making a certain number of these objects, and so forth. The description given here of a cost-accounting system is necessarily oversimplified, but the basic principles and concepts are the same.

Suppose that each procurement of supplies and parts, each object completed, the time consumed in each job, etc., were recorded by part number, workman's name, job number, date, and so forth, at the time the action was taken or the job completed; suppose that all this information for 1 week could be compiled on a single magnetic tape. The problem then reduces to searching, sorting, and ordering the items on the tape to answer the above questions. In general a computer that can perform these operations need be capable only of comparing, counting, and accumulating. Whether or not such a special-purpose computer is feasible depends on its cost and on the time it takes to process the necessary number of data.

A Small Retrieval Computer. Let us suppose that any item can be recorded in 50 characters or less (a character being 6 bits). Then the computer will first read a set of 50 characters into the temporary storage, a nonending loop of tape (see Fig. 4-8). Selected characters will be appropriately compared with characters from the previously loaded argument storage, a second loop of tape. Depending on the outcome of this comparison, the computer may then add into a partial sum in the accumulator some number represented by certain of the 50 characters, or may increase the count of the counter, and so forth. Actually, several sets of data may be recorded in the temporary storage unit. When writing back onto the magnetic tape, the computer can transpose sets if it is ordering the information, or can insert new information, and so forth. In summary,

successive items from the magnetic tape are stored temporarily in the computer; here information is extracted from them, or they are otherwise processed; then they are replaced on the magnetic tape, perhaps in some

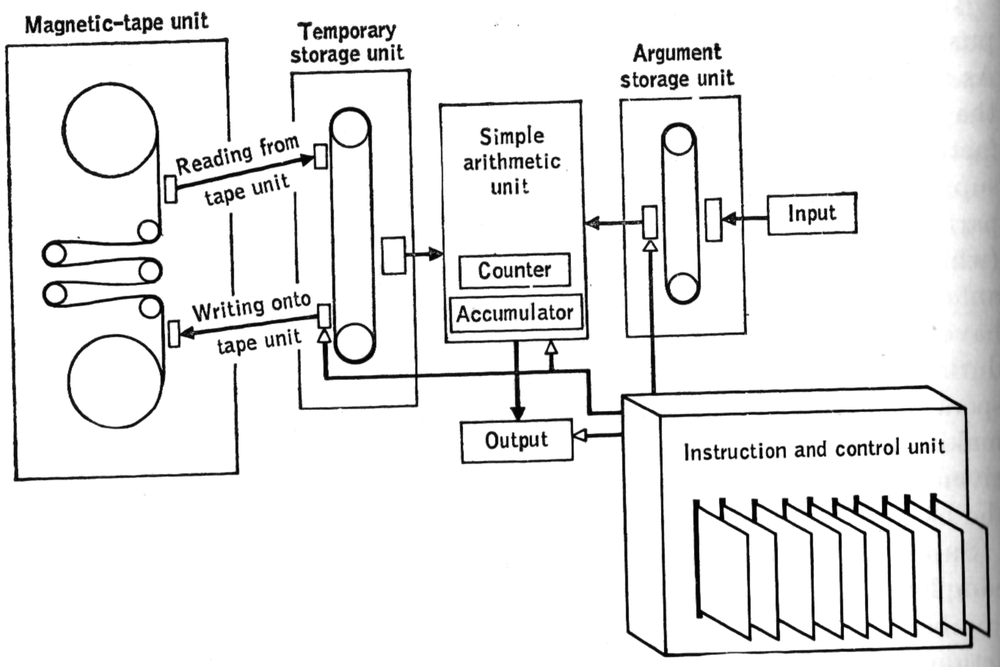


FIG. 4-8. Small retrieval computer.

transposed order. The processing is done in conjunction with information preloaded on the argument storage unit.

The number of instructions to be written into a program will be limited to the number of instructions that can be executed from the time an item is read from the tape until it is put back again. The instruction system can be very simple. Suppose that the instructions were punched on cards and the cards for one program put into slots in the control unit, where the holes on the cards can be sensed. The arguments on the argument-storage-unit tape loop, which are fixed during any program, can also be addressed. The instruction format could be as shown in Fig. 4-9. Each instruction is concerned with selected characters, as recorded on the cards, of the current item in the temporary storage unit. An

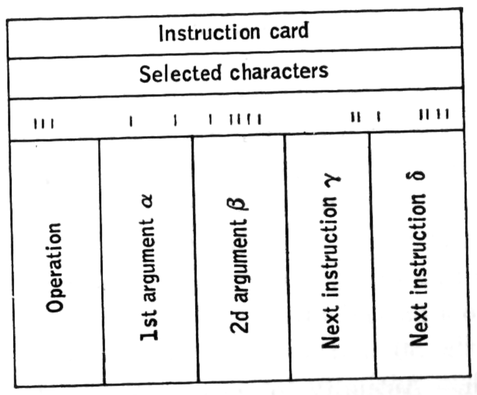


FIG. 4-9. Instruction format.

instruction may involve two of the arguments of the argument storage unit if, for example, a search for characteristics less than α but greater than β is desired. A two-way exit, γ or δ depending on the outcome of the comparisons, is allowed for flexibility in coding. Consider as an illustration the computation of the number of hours in a week that workman 1224 required to make part 2746 and the number of parts he completed in that week. Suppose that 40 were the code for an item recording a finished job. The instructions would compare the properly selected characters of each item with 1224, and 2746, and 40. When an "equal" comparison was made, the computer would add 1 to the counter and add the number of hours recorded on this item to the accumulator. If an equal comparison were not made, the computer would just replace the item on the tape. By this means the total number of man-hours worked in making a particular object during any specified time period can be determined in one pass through the tape. Similarly the number of man-hours that were worked by a certain person, or in a certain shop, etc., or the number of a certain part that were consumed, or the amount of a certain material that was used, in any shop or by any person, in making a particular object, during a specified time, can be determined in a single pass, and so forth. Suppose that the magnetic tape runs at a rate of 100,000 bits/sec, i.e., about 17,000 characters/sec (6 bits = 1 character), or 340 items/sec (50 characters = 1 item), or 20,000 items/min. Suppose also that there are 500 workmen in the factory, each generating an average of 5 items/day, or a total of 2,500 items/day, or 50,000 items/month (20 working days = 1 month). One pass through the tape containing 1 month's items would take approximately 2 min and 30 sec. Comparison with more conventional bookkeeping methods is hardly necessary. On the other hand, if we were to run a standard payroll calculation on the man-hours worked for each of the 500 men, it would take (500 men)(2 min 30 sec) $\cong$ 20 hr. Also, we have not mentioned the input and output equipment problem, which in the case of payroll, for instance, is a subject in itself.

Integration with a General-purpose Computer. In order to increase the working speed of this type of computer, we can enlarge the argument storage unit and include more accumulators and counters to accommodate more than one comparison, so that in one pass several pieces of information can be accumulated; or we can run several tapes in parallel, and so forth. The time per information pass can thus be decreased by a factor of 10 or 100, that is, to within 15, or even 1.5, sec. This would increase the complexity and hence the cost of the computer. Finally we can attach our special-purpose searching, sorting, and ordering computer to a general-purpose computer, enabling more complicated processing to be accomplished as well as more automatic operation (see Fig. 4-10). In

such a case the tapes can be searched at the same time as the general-purpose computer computes other things. The fixed card memory of the special-purpose computer will be replaced by a part of the high-speed memory shared with the general-purpose computer. Two codes will be written, one for the computing unit of the general-purpose computer, the other for the simple arithmetic unit of the special-purpose computer

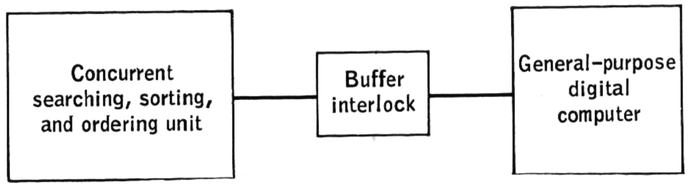


FIG. 4-10. Special-purpose computer attached to general-purpose computer.

An interlocking buffer must be designed so that the general-purpose computer will stop and wait if it needs some information not yet retrieved from the special-purpose auxiliary computer, and so that the searching computer will stop and wait if it in turn needs information not yet generated by the general-purpose computer. An auxiliary computer for this special purpose can be exceedingly effective and timesaving when large amounts of this kind of nonnumerical manipulative computing are necessary.

4-5. Manipulations with Rectangular Arrays: A Business and Logistics Computer†

The Need. An important class of computations occurring frequently in business and in logistics (i.e., processes involved in supply) is concerned with rectangular, matrixlike arrays of variables. The operations to be performed on the variables are usually very simple, the main problem being the manipulations necessary for handling such large-scale arrays. In this section we shall first indicate some applications of these techniques; we shall then show how a few relatively simple special-purpose instructions can significantly aid these processes; and finally we shall discuss a special-purpose computer especially designed to perform manipulations with rectangular arrays.

As a first example, consider the keeping and updating of a spare-parts inventory. Suppose for simplicity that there are 10 parts under consideration and that the first row of the array in Table 4-4 represents the desired inventory for each of the 10 parts—i.e., there *should be* 100 each of the parts in stock. The second row represents the actual stock level of

† The main concepts discussed in this section are based on the original research of Dr. W. H. Marlow, principal investigator of The George Washington University Naval Logistics Research Project.

each of the parts at time t. Now let us suppose that the parts are consumed in the course of three kinds of overhauls and that the number of parts used for one of each kind of overhaul appears on lines 4, 5, and 6, respectively. Now suppose that since time t there were one overhaul of kind 1, two overhauls of kind 2, and three overhauls of kind 3. The problem is to determine the present stock levels, i.e., to adjust row 2, and to determine how many of each part should be ordered in order to have the desired number of spare parts on hand, i.e., to determine row 3. To take account in the inventory levels of the parts used for the overhaul of kind 1, subtract row 4 from row 2, column by column; for the two

TABLE 4-4. DATA ARRAY FOR INVENTORY-UPDATING PROBLEM

	Part 1	Part 2	Part 3	Part 4	Part 5	Part 6	Part 7	Part 8	Part 9	Part 10
Row 1, desired inventory	100	100	100	100	100	100	100	100	100	100
Row 2, stock on hand (time t)	80	75	90	80	95	90	85	95	70	75
Row 3, parts to order										
Row 4, overhaul kind 1, parts used	5	15	5	10	5	0	0	5	10	5
Row 5, overhaul kind 2, parts used	10	0	0	10	10	20	15	5	10	0
Row 6, overhaul kind 3, parts used	5	0	10	10	5	10	15	20	0	5

overhauls of kind 2 multiply each figure in row 5 by 2, and subtract the result from row 2, column by column; and similarly form

$$\text{Row } 2 - 3 \times (\text{row } 6)$$

We thus obtain the present stock levels in row 2:

40 60 55 20 55 20 10 20 40 55

Subtracting row 2 from row 1 and putting the results in row 3, column by column, we obtain the number of parts to order of each type. Of course many other related problems can be similarly solved; e.g., we may record the usage of parts in each time period and then extrapolate column by column to predict usages in the future, and so forth. We have used only 10 different parts, but in realistic cases thousands of parts may be involved.

Another example is the calculation of building-material purchase schedules and requirements as functions of time, as would be needed in the construction of a ship or of a large office building. An array can be made of the materials needed at the construction site as a function of time. That is, successive rows of the array will represent successive

times, and the columns of the array will represent the various building materials; then an element in the array will indicate how much of a certain building material is needed at the building site at a particular time. However, there is a certain time lag that must be allowed for the delivery of each of the materials from the time it is ordered. This time lag differs for different materials and can be represented as a row of the array. The problem here is to schedule the orders so that the materials will arrive at the site at the proper time. In addition the costs of the materials per unit can be represented in another row. If the materials are paid for at the time they are received, then a calculation should be made as to how much money will be spent during each time period. This calculation can be made after the ordering schedules are set by multiplying the number of units to be received in a time period (i.e., a row) by the cost of each per unit and summing over the elements of the resulting row, i.e., adding the columns of that row independently.

The keeping of bank accounts presents similar array-manipulation problems. Here suppose that each row contains the balance of each of the savings accounts for successive weeks and that interest is computed monthly, based on the minimum weekly balance. Here the minimum of the rows each month must be obtained and the interest rate applied to each account (i.e., each column of the minimum record row). If the bank desired to know the total deposits each week, the row for each week is totaled. In this way calculations of interest, principal, penalties, interest rates, and other banking problems can be made.

Military and nonmilitary dynamic-operations simulation studies can involve these processes. For example, consider air-traffic control (in three dimensions). Three rows can record the x and y coordinates and altitudes, respectively, of the aircraft represented by the successive columns. If the x and y velocity components are known, the coordinates of the aircraft can be predicted in time and aircraft that may be flying too close appropriately warned. When changes in course are made, recalculations of predicted positions of all the aircraft become easy.

Of course, we could describe above only the simplest of applications. Our intention is to stimulate the reader's awareness of the great variety of fields in which problems arise that involve the manipulations of large arrays. Although a general-purpose computer can handle such problems, special-purpose computers can fulfill the need much more effectively and efficiently. We shall now describe some special-purpose operations that would be incorporated into such a computer.

Basic Operations and Instruction Format. With the above examples in mind the reasons for the definitions of the following basic manipulatory operations are clear. In fact it is quite surprising that these few simple operations can accomplish the task so well. The instructions that we

shall illustrate contain the necessary ingredients, although their form in actual practice may vary greatly. It is felt that a concrete example may best serve to demonstrate the concepts.

Since we are concerned primarily with arrays of numbers, i.e., with numbers arranged in rows and columns, suppose that we literally arrange these numbers in such an array in the computer's memory. Figure 4-11

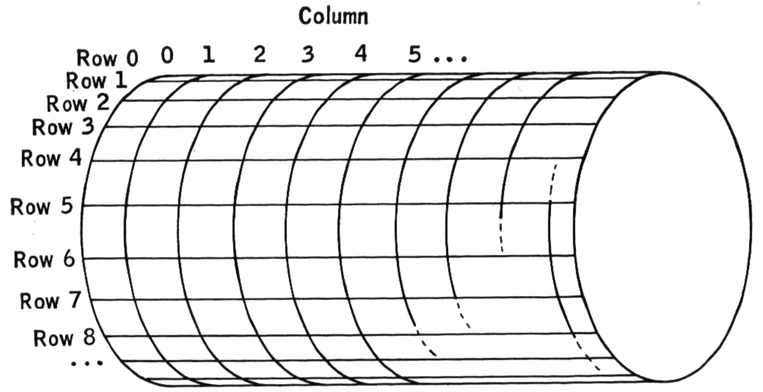


FIG. 4-11. Array of numbers stored on drum memory.

represents such an arrangement on a drum memory. Then in order to "address" a particular number, all we need specify is its *row* and *column*. For reasons that will presently appear clear, we denote the number in row j and column x by the following functional notation: $f_j(x)$—see, for example, Table 4-5.

The address of $f_j(x)$ is thus specified by the *pair of indices: j, x*. We use the phrase *generalized address* because the instruction actually indicates a collection of addresses in the manner now to be described. Consider a multiplication instruction. Rather than form the product of

TABLE 4-5. NOTATION OF THE ARRAY

	Column x								
	0	1	2	· · ·	p	· · ·	q	· · ·	n
0	$f_0(0)$	$f_0(1)$	$f_0(2)$	· · ·	$f_0(p)$	· · ·	$f_0(q)$	· · ·	$f_0(n)$
1	$f_1(0)$	$f_1(1)$	$f_1(2)$	· · ·	$f_1(p)$	· · ·	$f_1(q)$	· · ·	$f_1(n)$
2	$f_2(0)$	$f_2(1)$	$f_2(2)$	· · ·	$f_2(p)$	· · ·	$f_2(q)$	· · ·	$f_2(n)$
· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·
i	$f_i(0)$	$f_i(1)$	$f_i(2)$	· · ·	$f_i(p)$	· ·	$f_i(q)$	· · ·	$f_i(n)$
· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·
j	$f_j(0)$	$f_j(1)$	$f_j(2)$	· · ·	$f_j(p)$	· · ·	$f_j(q)$	· · ·	$f_j(n)$
· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·
m	$f_m(0)$	$f_m(1)$	$f_m(2)$	· · ·	$f_m(p)$	· · ·	$f_m(q)$	· · ·	$f_m(n)$

just two numbers, $f_i(x) \cdot f_j(x)$, we wish now, *with a single instruction,* to form a whole row of successive column-by-column products, that is, $f_i(1) \cdot f_j(1)$, and $f_i(2) \cdot f_j(2)$, and $f_i(3) \cdot f_j(3)$, and so forth. We might represent this as the operation $f_i(x) \cdot f_j(x)$ for $x = 1, 2, 3, \ldots$. To add a little more flexibility to the operation, we might form, with a single instruction, $f_i(x) \cdot f_j(x)$ for $p \leq x \leq q$, that is, the successive products for only the columns p through q. For example, if $i = 1, j = 2, p = 5$, and $q = 8$ we would form (see Table 4-6)

$$f_1(5) \cdot f_2(5) \qquad f_1(6) \cdot f_2(6) \qquad f_1(7) \cdot f_2(7) \qquad f_1(8) \cdot f_2(8)$$

Thus the multiplication instruction must now contain the row indices i and j and the column indices p and q; this collection of indices represents the so-called "generalized addresses." Suppose that we wish the results to be placed in row k of the corresponding columns p to q; then the index k must also appear in the instruction. Thus i, j, and k, together with p and q, represent the generalized α, β, γ, and δ addresses, and all these indices must appear in the instruction format itself.

TABLE 4-6. ILLUSTRATION OF OPERATION $f_1(x) \cdot f_2(x)$, $5 \leq x \leq 8$

	Column									
	0	1	2	3	4	5	6	7	8	9
Row 1.....	9	7	8	6	1	5	2	4	3	1
Row 2.....	6	2	5	1	0	3	4	8	7	2
Row 3.....	...	...	...	...	...	15	8	32	21	...

A description of the fourth generalized address, that of the next instruction, requires reconsideration of the drum memory. An instruction would appear as a partial row of numbers and can be recorded along with the data on the drum. For greatest efficiency a row length on the drum should be a multiple of an instruction length. To address the part of a row that is an instruction, only the row and column of the first number of the instruction need be known, since the length of the instruction is fixed by the instruction format (see below). Let us denote the row and column number of the first characteristics of an instruction by δ_r and δ_c, respectively. These must also appear in each instruction word.

There are several other considerations that can lead to greater flexibility in our instruction format. First suppose that we desired to form $C \cdot f_i(x) \cdot f_j(x)$ (for $p \leq x \leq q$), where C is a constant. Since this is frequently desired (e.g., interest and discount rates), it is convenient to include the constant C in the instruction format itself, rather than

address it separately. Second, observe that so far we can work on the different rows *within* the same column but we cannot have a number in one column affect a number in another column. Thus, for example, the product $f_i(x) \cdot f_j(x)$ becomes $f_k(x)$ in the same column. To circumvent such a restriction, it suffices to introduce into the instruction format the alternative of placing the result of the operation in corresponding columns in the kth row, *or of placing the result one column to the left in the kth row, i.e., displaced one column.* That is, one may choose to have the product $f_i(x) \cdot f_j(x)$ become $f_k(x)$ or $f_k(x - 1)$. Finally, recall that (γ) is usually *replaced* by the result of the operation. However, great flexibility results if there is introduced into the instruction the alternative of either replacing (γ) *or else accumulating* (*adding*) *the results of the operation to the original contents of* γ. As an example, suppose that, given the initial rows 1, 2, and 3 of Table 4-7, we formed $f_1(x) \cdot f_2(x)$, for $5 \leq x \leq 8$ (that is, $i = 1, j = 2, p = 5, q = 8$), and *accumulated* the result in row 3, *displaced.* Then we would have

$$f_3(x - 1) \text{ final} = f_1(x) \cdot f_2(x) + f_3(x - 1) \text{ initial}$$

for $5 \leq x \leq 8$ (see Table 4-7).

TABLE 4-7. ILLUSTRATION OF OPERATION $f_1(x) \cdot f_2(x)$, $5 \leq x \leq 8$, PLACED IN ROW 3 DISPLACED BY ONE COLUMN, AND SO AS TO ACCUMULATE

	Column									
	0	1	2	3	4	5	6	7	8	9
Row 1.....	9	7	8	6	1	5	2	4	3	1
Row 2.....	6	2	5	1	0	3	4	8	7	2
Row 3, initial....	3	2	4	6	2	1	5	3	2	6
Row 3, final.....	3	2	4	6	17	9	37	24	2	6

With these considerations in mind we can represent an instruction format as

Operation code	Row i	Row j	Row k	p	q	Displaced or not	Accumulated or not	C	δ_r	δ_c

We shall use the convention that 1 means *displace*, 0 means *do not*, and that 1 means *accumulate*, 0 means *do not*, in their respective columns.

The instruction that will produce the final result shown in Table 4-7 is

Operation	i	j	k	p	q	Displace	Accumulate	C	δ_r	δ_c
Multiply	1	2	3	5	8	1	1	1		

We are now ready to define seven basic operations, where the column-by-column disposition of each type of result is understood to be in accord with the above discussion, and where only the column-by-column operations on the arguments need be given.

1. Multiplication
$$Cf_i(x) \cdot f_j(x) \qquad p \le x \le q$$
2. Division
$$\frac{Cf_i(x)}{f_j(x)} \qquad p \le x \le q$$
3. Linear combination
$$Cf_i(x) + f_j(x) \qquad p \le x \le q$$
4. Minimum
$$\min_{i,j}[f_i(x), f_j(x)] \qquad p \le x \le q$$

The final three instructions require some additional explanation:

5. Displaced linear combination $Cf_i(x - 1) + f_j(x)$. Here the addition is not performed column by column, but rather *displaced* column by column. Of course, it is always understood that $p \le x \le q$, but then, when $x = p$ and $p = 0$, the meaning of $f_i(0 - 1)$ is undetermined. We therefore define $f_i(0 - 1) = f_i(0)$.

6. Partial scalar product $\sum_{\mu=p}^{x} C \cdot f_i(\mu) \cdot f_j(\mu)$. This defines *one result for each* x such that $p \le x \le q$. For example, if row i and row j were

	Column					
	0	1	2	3	4	5
Row i.....	7	6	5	4	3	2
Row j.....	2	3	4	5	6	7

and $p = 2$ and $q = 5$, then the results would be

$5 \times 4 = 20$ $\quad 5 \times 4 + 4 \times 5 = 40$ $\quad 5 \times 4 + 4 \times 5 + 3 \times 6 = 58$

and $\quad 5 \times 4 + 4 \times 5 + 3 \times 6 + 2 \times 7 = 72$

7. Compare $(\min_{p \le x \le q}[f_i(x)]) : C$. This operation does not yield a column-by-column result; rather it is a *two-way exit.* If the minimum number in row i, columns p to q, is *greater than* C, then the next instruction is to be taken as the instruction that has row j, column k as the address of its first number; otherwise, i.e., if $\min_{p \le x \le q}[f_i(x)] \le C$, the next instruction is to be chosen as usual, from δ_r, δ_c. Of course *displace* and *accumulate* have no meaning for a *compare* instruction.

Illustrations of Coding. As a first illustration of using these special-purpose instructions in writing codes, consider the problem of accumulating the amounts within each column, from row i_0 to row $i_0 + h$, putting

TABLE 4-8. CODE TO ACCUMULATE THE ROWS i_0 TO $i_0 + h$, COLUMN BY COLUMN†

Instruction address	Operation	i	j	k	p	q	Displace	Accumulate	C	δ_r	δ_s	Remarks
0, 0	3	i_0	$i_0 + 1$	$i_0 + 1$	0	21	0	0	1	1	0	Add i to $i + 1$; put in $i + 1$
1, 0	7	0	3	0	1	1	...	...	$i_0 + h - 1$	2	0	i:$i_0 + h - 1$ $>$: Stop iteration $\le$: Continue
2, 0	3	4	0	0	1	3	0	0	1	0	0	Update (0,0)
3, 0												Stop computer
4, 0	0	1	1	1	0	0	0	0	0	0	0	Constant

† Recall that the columns are numbered 0 through 21.

the results into row $i_0 + h$. Suppose for simplicity that the drum has 22 columns (just twice the instruction length) and that the code is written in addresses 0, 0; 1, 0; 2, 0; etc. The code will make use of operation 3, with $C = 1$, *not* accumulating the results of summing i and $i + 1$. (Why?) Then a *compare* instruction will occur for the tally, and the iterations will proceed by adding 1 to the generalized addresses i, j, and k of the first instruction until the accumulation is completed. Since the first instruction is updated on each iteration, its i address can be used as the tally. The code appears in Table 4-8.

As a second example, consider a linear extrapolation. Suppose that each column represented an aircraft and that rows x_1, y_1, and t_1 represented the x and y coordinates of each aircraft at time t_1, and rows x_2, y_2, and t_2 represented the x and y coordinates of each aircraft at time t_2. The problem is to make a linear extrapolation of the coordinates of each plane at time t. The formulas to be used are

$$x = (t - t_2)\frac{x_2 - x_1}{t_2 - t_1} + x_2 \qquad \text{and} \qquad y = (t - t_2)\frac{y_2 - y_1}{t_2 - t_1} + y_2$$

We first form $x_2 - x_1$ in row x_1, $y_2 - y_1$ in row y_1, $t_2 - t_1$ in row t_1, and finally $t - t_2$ in row t. Then we form the quotient of rows x_1 and t_1 in row x_1 and of rows y_1 and t_1 in row y_1. Next we form products of rows t and x_1 in row x_1 and of rows t and y_1 in row y_1. Finally row x_2 is added to row x_1 and put in row x_3; and row y_2 is added to row y_1 and put in row y_3; and the extrapolation is completed. The method of coding is straightforward, using $C = -1$ in operation 3 for subtraction.

As a third example, consider a numerical integration according to the rectangular approximation. Suppose that the successive values of the function at points x are given by $f_i(x)$ and that the intervals are unequal and are given in row $i + 1$. Then for the rectangular approximation we want to form the product of rows i and $i + 1$, column by column, and then accumulate this product over the columns. Here we make direct use of operation 6 and form

$$\sum_{u=0}^{21} f_i(u) f_{i+1}(u)$$

in row $i + 2$. Then the value of the approximation is found in row $i + 2$, column 21.

Description of a Special-purpose Business or Logistics Computer. One of the most important features that a special-purpose computer of this type should have is rapid, flexible input-output. A general characteristic of the problems under consideration in this section is the large mass of input data to fill the array and the large mass of data computed from the array. This is in fact the main reason for having instructions that operate with generalized addresses. The given data may be characterized as items and characteristics associated with the items. Here the characteristics are the numbers upon which it is necessary to perform operations, and the items are the specifications of these numbers. Hence on the drum the *location of a number* can represent the item or specification, while the *number itself*, i.e., the contents of this location, will be the *characteristics.*

For example, in a naval supply problem an item might consist of the set of numbers (x_0,x_1,x_2,x_3), where x_0 = ship number, x_1 = time period (e.g., month and year), x_2 = Navy material class (e.g., class 75, diesel engine parts), and x_3 = source of supply (e.g., the vessel itself, an afloat activity, a shore activity). The associated characteristics might consist of the set of numbers (y_0,y_1,y_2), where y_0 = total number of items consumed, y_1 = total weight in pounds, and y_2 = total volume in cubic feet. The source of this data may have covered 12 ships, during 72 time periods, for 50 Navy material classes, and from three sources of supply, or 129,600 items. For each of these items there appear three

six-digit numbers representing y_0, y_1, and y_2. The data would then contain 2,332,800 digits.

Processing this data might consist in computing and printing tables with entries y_0, y_1, and y_2 corresponding to some specified set of the x's. For example, the set (x_0,x_1) would lead to a two-way table of vessel number vs. time period, with entries covering all classes of material from all sources. Or additional functions of the y's may be desired. Average values are about the simplest example: $y_3 = y_1/y_0$ = average weight in pounds; $y_4 = y_2/y_0$ = average volume in cubic feet; $y_5 = 2{,}240y_2/y_1$ = stowage factor in cubic feet per long ton.

It often occurs that a characteristic of one problem becomes an item specification for another problem, or the reverse. For instance, it might be desired to compute a table giving the values of y_0, y_1, and y_2 for each x_0, x_1, and y_3, that is, for each ship, time period, and *average weight.* In such a case the numbers y_0, y_1, and y_2 will have to be relocated so that x_0, x_1, and y_3 will now specify their locations. Hence it becomes clear that in business and logistics computations searching, sorting, and ordering play an important role, in addition to the types of computations considered above.

In the previous section we assumed that the information was recorded on a magnetic tape; here, however, we shall consider searching, sorting, and ordering of items by characteristics accomplished on an auxiliary drum, different from the one with which the computations are performed. The input data, which might, for instance, be on punched cards, is read onto the searching drum, both the items and the characteristics being recorded. This drum is appropriately searched, and the characteristics determined by the search are recorded on the computational-array drum in positions (i.e., addresses) that specify the associated item. Any rearrangement or reordering is accomplished by use of the searching drum.

Our special-purpose business or logistics computer is thus seen to be *two interlocking computers,* an *array-manipulation computer* and a searching or *information-retrieval computer* (see Fig. 4-12). Each would have its own programming system. The array-manipulation computer would have special-purpose array-manipulation instructions as described above; the information-retrieval computer would have instructions enabling it to perform the operations described in the previous section. Ideally the two computers should be able to operate concurrently; appropriate interlocking conditions would have to be satisfied, so that the two computers and computer programs would be properly synchronized.

In addition, we could attach a general-purpose computer to this setup to have a large-scale system (Fig. 4-13). Again, for concurrent opera-

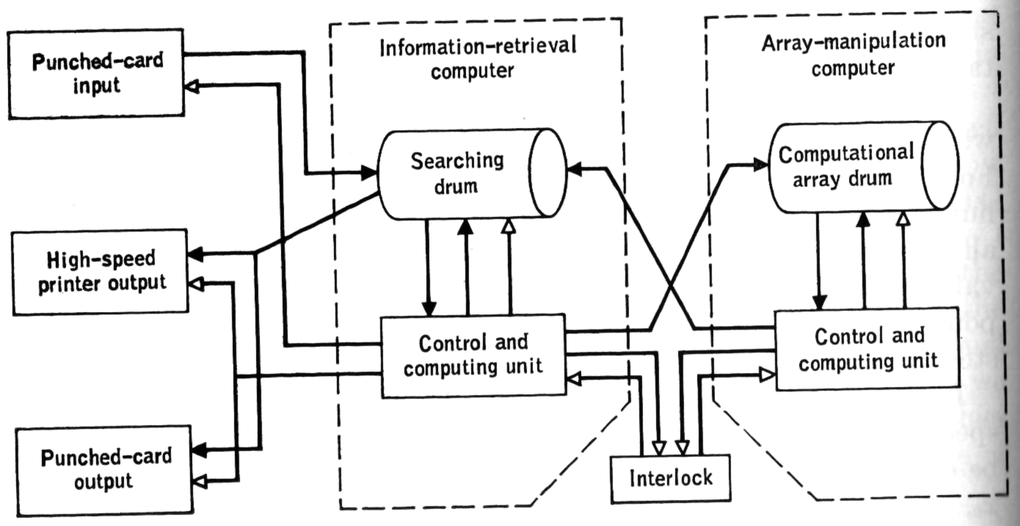


FIG. 4-12. Special-purpose business or logistics computer.

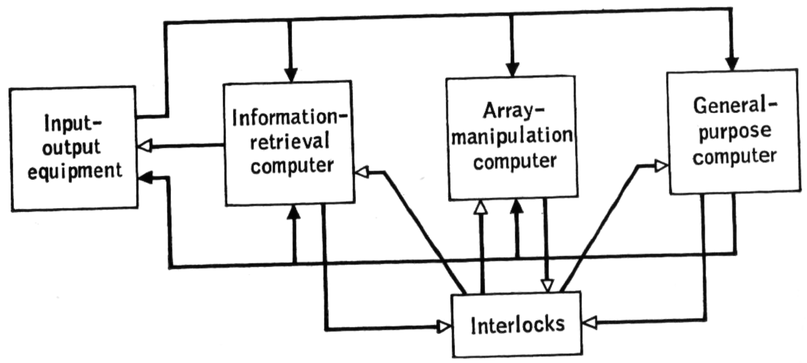


FIG. 4-13. Large-scale system consisting of information-retrieval, array-manipulation, and general-purpose computers.

tion interlocks must be maintained. More will be said about such an arrangement in the next section.

4-6. The Large-scale Data Processor

Special Input-Output Problems. When data is collected, the recording scheme used is ordinarily determined from criteria dictated by the particular problem, by convenience, etc. Rarely is the best form for computer utilization considered as a factor. Also, data produced by a computer will not in general be directly compatible for input into another type of computer. These problems become extremely serious in, for example, missile-testing programs, aircraft-flight evaluations, etc., where exceedingly large masses of data are produced very rapidly. The problem of conversion of this data to a form that can be read directly into the computer for analysis then becomes almost a larger task than collecting the data.

One method for solving this problem is to have a special input drum for the computer. Everything that is read into the computer is *read directly onto this drum just as it appears on the input medium.* If the input were punched cards with 80 columns and 10 rows, then the data of the cards would be read onto the drum just as it appears on the cards, with a unit corresponding to a hole, a zero to no hole. Then a special *translating computing unit* would be coded to interpret this information appropriately. The translating unit would interpret the raw information on the drum, change it into appropriately coded binary data, and put it into the computer's main memory. This technique is extremely flexible, since all that is needed is a method for getting the raw data onto the drum, in any digital fashion whatsoever.

This naturally brings up a second problem. Since reading in and translating the data will be time-consuming, it would be more efficient if the computer could be computing on other data, or even another program, concurrently with the read-in and translation. Such a *concurrent-input feature* presents no problems other than logical and electronic design, provided that certain interlocks are observed. In reading data into a computer it is certainly known in which part of the main memory the information will eventually be stored. A simple type of interlock will halt a concurrent computer program that refers to a main memory address that has not yet been read into; when the input information is inserted into this address, the program automatically continues.

In so far as output from the computer is concerned, the translator acts in reverse, putting information on the special drum in a form which when read out from that drum will correspond to any desired external coding scheme. Similarly, with proper interlocks concurrent output can be accomplished.

The Large-scale Data Processor. The large-scale data processor as described in this paragraph (see Fig. 4-14) is a fictitious computer; its description is used here as a vehicle for discussing the problems that arise in *multiple-unit system.* The purpose of this illustrative computer would be to perform general data processing. It encompasses each of the three features described in Secs. 4-3, 4-4, and 4-5, in addition to the translating component. Hence it is comprised of a general-purpose computing unit, a logical-control computing unit, a searching, sorting, and ordering computing unit, an array-manipulating computing unit, and an input-output translator computing unit. It contains a high-speed memory, a general-purpose magnetic drum, a translator input-output drum, a sorting drum, an array-manipulation drum, as well as magnetic-tape units. It has conventional input-output equipment and a signal-sensing input unit for real-time control.

The main point, however, to be illustrated by this example is that any

or all of the computing units may be operating (i.e., computing) concurrently, using any or all of the memories or input-output units. The mechanisms that enable such concurrent operation are found in the in-out selector–concurrent input-output interlock control and in the computing-units interlock control.

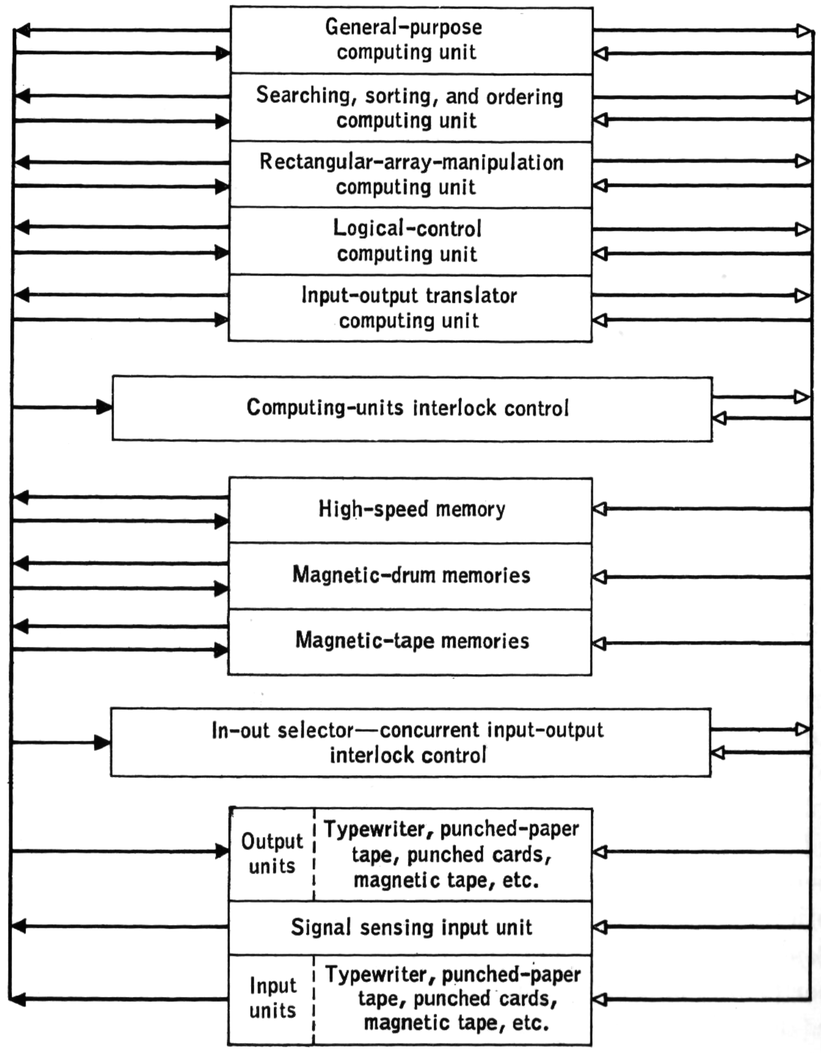


FIG. 4-14. Large-scale data processor.

Let us consider the computing-units interlock control first and describe two possible ways it can operate. Consider the case where our data processor is operating on multiple independent codes simultaneously. The instructions for each of these codes are found in the memory units. At any time, each code is being computed in a different computing unit. Each computing unit has its own instruction register, and its own current-address register that sequences the program according to the instruction definitions of the particular unit in which it is located.

There are times, however, when one code will want to operate in a different unit, will want to change computing units. These *change-unit* instructions are not handled in the computing units; instead they are sensed and sent to the computing-units interlock control. This interlock control keeps track of which units are being used. If a code wants to change to a computing unit that is being used, the code progress is stopped until the desired unit is free. A waiting line, or queue, is formed if necessary for each computing unit. The order, or priority, in which waiting codes are selected from the queues can be made as complicated as is considered necessary for the purposes of the computer design.

When the desired computing unit becomes available, the contents of the instruction address register is transmitted to this computing unit and the code is continued in its new unit. In other words, a computing unit processes the instructions it retrieves from the memory as directed by its current-address register; when changing from one code to another, a computing unit merely needs to have its current-address register changed. The control mechanism for this change is located in the computing-units interlock control. By this means each of the computing units can act almost as a separate computer in its own right, except when required to idle by the computing-units interlock control. Even though this concept for handling multiple codes at the same time is reasonably simple, the detailed techniques in a specific case may present many special considerations and exceptions.

The second way in which a computing-unit interlock control might operate is to have only a single code being computed. However, it often happens that different parts of this code can be computed at the same time. That is, the logic or flow diagram of the code may not necessitate a completely serial sequencing of operations, but several parts of the code might be processable in parallel at the same time. In this mode the computing-units interlock control scans the entire code, recognizing parts of the code that can be executed independently. It does this by observing memory reference locations. If one part of a code never refers to a memory location used by another part of the code, then it can be accomplished at the same time as that other part of the code. The rules for observing this can become rather complicated. The computing-units interlock control must accomplish both functions; how thoroughly it accomplishes either function depends on the systems design and purpose of the computer.

Finally we consider the in-out selector–concurrent input-output interlock control. Besides in-out selections, this control unit must accomplish input-output interlock control similar to that described above. All input-output instructions are automatically sensed and sent to this interlock control. If free, the appropriate unit is selected; if not, the code will wait, as was the case in the computing-units interlock control.

This interlock control must communicate with the computing-units control, so that a code may be stopped when it wants to use a memory location not yet loaded. Again the details can become exceedingly complicated.

4-7. Exercises

For the digital differential analyzer:

4-1. Compute sin x and cos x, and graph the results as they would be computed (see, for example, Fig. 4-1). If the word length were 4 bits, how would the problem be scaled?

4-2. Draw the arrangement of units to compute $y = 1/x$.

4-3. Draw the arrangement of units to compute $y = \ln x$. [HINT: $dy = (1/x)\,dx$. Let $w = 1/x$; then $dy = w\,dx$, and $dw = (-1/x^2)\,dx = -w\,dy$.]

4-4. Draw the arrangement of units to compute $y = x^n$. {HINT: $dy = nx^{n-1}\,dx = x^n[(n/x)\,dx] = x^n\,d(n \ln x) = y\,d(n \ln x)$.}

4-5. Draw the arrangement of units to solve the following simultaneous differential equations:

$$\frac{dy_1}{dx} = \lambda_{12}y_2 + \lambda_{13}y_3 \qquad \frac{dy_2}{dx} = \lambda_{21}y_1 + \lambda_{23}y_3 \qquad \frac{dy_3}{dx} = \lambda_{31}y_1 + \lambda_{32}y_2$$

4-6. How can a digital differential analyzer be used in conjunction with machine-tool control (see Chap. 1)?

For the real-time control computer:

4-7. How would the external-control instruction break the loop when *no* comparison is successful? (HINT: A dummy comparison would automatically compare favorably. How would this dummy comparison be arranged, i.e., what would be the first and second words of the dummy condition?)

4-8. Suppose that the input word has the following inputs (see Fig. 4-15): tank 1 full, **L(1)**; tank 2 full, **L(2)**; valve 1 open, **V(1)**; valve 2 open, **V(2)**; valve 3 open, **V(3)**. For the output or control-signal word let **O(1)** mean open valve 1; **O(2)**, open valve 2; **O(3)**, open valve 3; **C(1)**, close valve 1; **C(2)**, close valve 2; and **C(3)**, close valve 3. Define an appropriate instruction, including an external-control

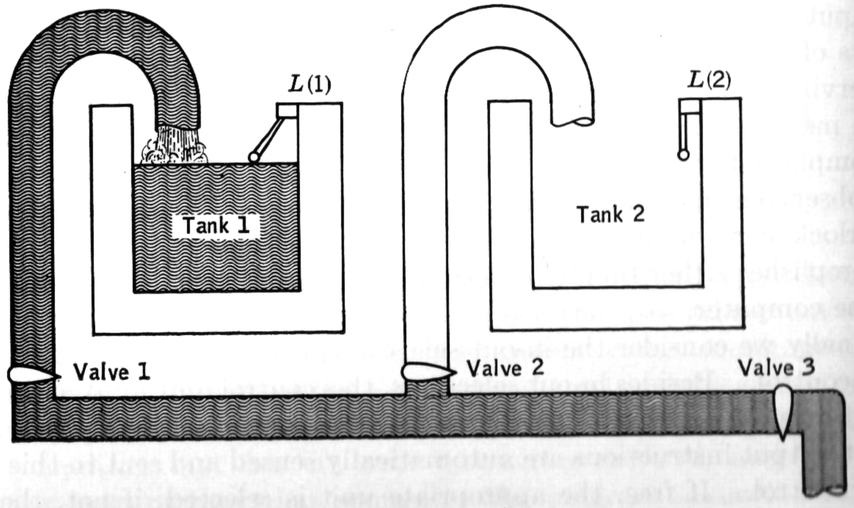


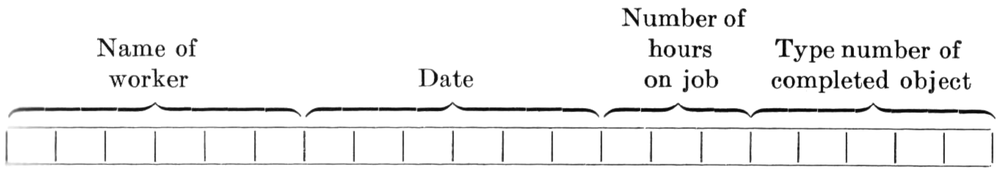
FIG. 4-15. Simple system controlled by real-time logical-control computer.

instruction, for the logical-control computer as discussed above, and code the process of filling the two tanks, one at a time.

4-9. Code the problem used in Exercise 4-8 without the use of an external-control instruction.

For the information-retrieval computer:

4-10. Suppose that a factory item is composed of the following word:



where each box can be filled with a digit or an alphabetic character. Define an instruction system for a special-purpose retrieval computer as shown in Fig. 4-9, and write a code that will determine how many objects of type 54321 were completed between Mar. 3, 1959, and Mar. 15, 1959.

For the business and logistics computer:

4-11. Consider the example of computing building-material purchase schedules and requirements. Suppose that the requirements at the site for each month during the construction are as follows:

Month	Type of material										
	0	1	2	3	4	5	6	7	8	9	10
March	100	75	60	54	0	0	20	60	10	20	30
April	50	0	40	60	0	0	20	90	10	20	0
May	25	0	30	50	0	0	30	150	10	30	0
June	30	115	20	70	0	0	50	50	50	0	20
July	0	0	10	30	10	20	90	20	70	0	10
August	0	100	5	50	50	30	100	10	90	50	5
September	0	0	0	90	100	100	150	0	100	90	0
October	0	0	0	20	0	150	200	0	150	100	0

Suppose that the delay time and the cost per unit are as follows (where the cost per unit of the material varies with the number of units ordered as shown):

	Type of material										
	0	1	2	3	4	5	6	7	8	9	10
Delay time, months	3	5	6	2	4	6	5	3	5	6	7
Units ordered:											
0–9	10	50	35	20	5	40	30	20	30	100	150
10–19	10	45	35	15	5	35	30	15	30	100	120
20–49	10	40	30	14	5	30	25	15	25	95	110
50–99	7	40	25	13	4	25	25	10	25	95	100
100 or over	7	38	20	12	3	20	20	10	20	90	100

Determine the schedule for the ordering, and determine how much money will be spent each month, assuming that the orders are paid for at the time they are received.

4-12. Write a code that will compute Exercise 4-11.

4-13. Write a code for numerical integration by the trapezoidal rule.

4-14. Write a code for linear extrapolation.

4-15. What situations might require interlocks between the information-retrieval computer and the array-manipulation computer?

For the large-scale data processor:

4-16. Draw a flow chart of a code that would interpret raw data from cards. Suppose that a computer word contains 15 bits, with the decimal point to the right. Assume that the raw data appear on cards with 80 columns and 10 rows, where each card is interpreted as containing 20 four-digit numbers and the digit associated with each column is the row location of a single punch in that column. For example, the first two numbers on the card illustrated in Fig. 4-16 are 1,340 and 1,235. Assume

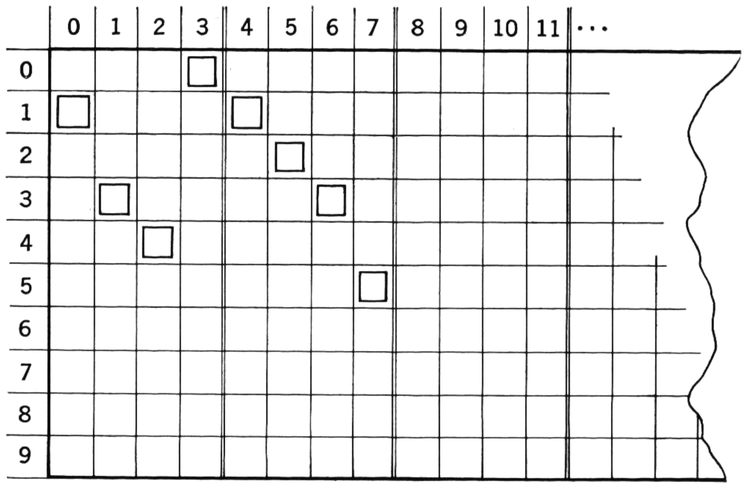


FIG. 4-16

that the translator drum records the *rows* of each card in sequence, from 0 to 9. The code is to compile the successive numbers (i.e., successive groups of four columns) *in binary* in the main memory of the computer.

REFERENCES

References on Digital Differential Analyzers. In general, digital differential analyzers are far more versatile and complicated than described in Sec. 4-2. For a more thorough understanding of their variations and potentialities see the following references:

Bush, Vannevar: Differential Analyzer, *J. Franklin Inst.*, vol. 212, no. 4, pp. 447–448, October, 1931.

Donan, J. F.: The Serial-memory Digital Differential Analyzer, *Mathematical Tables and Other Aids to Computation*, vol. 6, no. 38, pp. 102–112, April, 1952.

Hartree, Douglas R.: "Calculating Instruments and Machines," University of Illinois Press, Urbana, Ill., 1949.

Mendelson, Myron J.: The Decimal Digital Differential Analyzer, *Aeronaut. Eng. Rev.*, February, 1954, pp. 42–54.

Palevsky, M.: The Design of the Bendix Digital Differential Analyzer, *Proc. IRE*, vol. 41, pp. 1352–1356, October, 1953.

Silber, W. B.: Function Generation with a DDA, *Instr. Control Systems*, vol. 33, no. 11, pp. 1895–1899, November, 1960.

Sprague, R. E.: Fundamental Concepts of the Digital Differential Analyzer, *Mathematical Tables and Other Aids to Computation*, vol. 6, pp. 41–48, January, 1952.

Weiss, E.: Applications of the CRC 105 Decimal Digital Differential Analyzer, *IRE Trans. on Electronic Computers*, vol. EC-1, pp. 19–24, December, 1952.

Winslow, D. J.: Incremental Computers in Simulation, *Meeting of South East Simulation Council*, Oct. 30, 1958, Huntsville, Ala.

Multiple-component Computing Systems. In Sec. 4-6 the systems design of computers composed of several different interlocked computing components was described. The following references give further details about specific examples of systems designs, illustrating this principle:

Adams, C. W.: Trends in Design of Large Computer Systems, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 361–364.

Dreyfus, Philippe: France's Gamma 60—A Step Forward in Data Processing? *Datamation Research Eng. J.*, May–June, 1958, p. 34.

Dunwell, S. W.: Design Objectives for the IBM Stretch Computer, *Proc. Eastern Joint Computer Conf.*, December, 1956, p. 20.

Eckert, J. P.: Univac-Larc, the Next Step in Computer Design, *Proc. Eastern Joint Computer Conf.*, December, 1956, p. 20.

Examples of Computer-systems Control

Eisenhardt, R. D., and J. J. Williams: Closed Loop Computer Control at Luling, *Control Eng.*, vol. 7, pp. 103–114, November, 1960.

Marton, F. D.: Computer Control of Vinyl Chloride and Acrylonitrile Process, *Instr. Control Systems*, vol. 33, p. 2065, December, 1960.

PART 2

AUTOMATIC-PROGRAMMING LANGUAGES

CHAPTER 5

DEVELOPMENT OF AUTOMATIC PROGRAMMING

5-1. Introduction

Part 1 of the book was concerned with the details of programming and coding at the instruction, or *machine-language*, level. The four-, three-, two-, and one-address instruction formats were described and compared. The octal-coded binary and decimal-coded binary systems were compared. The steps in preparing a flow chart, symbolic code, and finally the code itself written in the actual machine language were described. Decision branches, instruction modification, and recursion codes were discussed and examples given. Coding was found to be detailed and laborious, and the one-address system particularly aggravated these qualities.

Advanced programming techniques attempt to lighten the load of the programmer and coder. Their purpose is to try to make the computer itself help prepare the program or code, minimizing the amount of writing a programmer need do. The means used to accomplish this is to prepare or precode the computer with an intermediate, or *automatic*, program which is always available in the computer memory. This intermediate or automatic code encompasses certain aspects of coding that are common to many programs and of general applicability. Then, when the programmer is coding any specific problem, his work can be shortened by taking advantage of already precoded items of the automatic program that are already in the computer. Such automatic programming can greatly simplify the coding of computers otherwise difficult to code. Unfortunately, however, the coder must then learn how to use not only the computer but also the automatic-programming languages.

Rarely is a program written that is found to be absolutely correct the first time it is run on the computer. Hence in this chapter we first consider automatic aids for analyzing an already coded program for errors. Next automatic programs designed to lighten the load of the programmer are considered. These can be roughly classed in two cate-

gories—the interpretive routines and the compiling routines. Various examples are discussed as illustrative of the methods and techniques of formulating and using such automatic programs. The specific examples given are intended to illustrate the principles involved, and their particular form as chosen for this chapter was dictated by pedagogical considerations only. As will become clear during the course of the chapter, there are many other ways to solve the problems posed, and in actual practice the final choice depends on the primary purpose of the code and the detailed characteristics of the computer to be used.

5-2. Program Debugging Methods and Routines

The Problem of Program Debugging. The first time a program is tried on a computer, it usually does not run correctly. If the start button is pushed and the program allowed to run, one of four results can occur: First, the computer might run up to a point and then just stop, or "hang up." Second, the computer could continue to run for an excessively long time, neither stopping nor producing any results. Third, the computer could run and, as expected, print out results, but wrong results. And finally the program could run correctly. In the first three cases an error is indicated in the program, and the problem is to detect where—at what instruction or instructions—the error occurs and then to correct it. This procedure is often referred to as "debugging" a program.

It is hard to describe in general how to debug a particular program; ingenuity on the part of the programmer, systematic and neat flow charts, coding sheets with detailed remarks, and a thorough knowledge of the code are, of course, all essential. When a program just stops, this usually indicates that the computer has been asked to do an illegal instruction; for instance, the instruction might have an operation code not corresponding to any operation or might contain an address that the computer does not have, etc. Illegal instructions are often the result of an error in instruction modification. When a program never stops, the program usually has a loop with a faulty exit. The exit may not have been properly set up, or else a nonconverging iteration may be in process, etc. The printing out of wrong results requires examination of these results in the light of the problem: Is the scaling wrong? Are the constants correct? Has any phase of the calculation been skipped; etc.?

However, certain techniques can be discussed that are often used to aid the debugging process. These can be generally classified into *dynamic* debugging and *post-mortem* debugging. Dynamic techniques try to check the program while it is actually running on the computer; post-mortem techniques examine the computer memory after the program has stopped, or "hung up."

Dynamic Program Debugging. Let us first consider dynamic program debugging. There are two generally used methods: the first is called the *break-point* (or *ckeck-point*) method; the second is called the *automonitor* method. The break-point method consists in writing into the program, at critical intermediate points, instructions that either will stop the computer so that important intermediate results can be observed or will make the computer print out such intermediate results automatically. The sign bit on an instruction word is often used to facilitate such a process. For example, some computers are designed so that after executing an instruction with a negative sign the computer will stop. Usually such computers also have the following additional feature: A manual switch on the control panel can be thrown so that in one position the computer will *not* stop after a negative instruction—this being called the normal switch position—and so that in the other position the computer will stop—this being called the break-point switch position. The advantage of this control-panel switch is clear: in debugging a program, the negative instruction stops are used, and the switch is set accordingly; when the program has been corrected, these negative stops are no longer desired and hence the switch is thrown to normal.

Another break-point procedure depends on the programmer's inserting into the program at intermediate critical points manual-switch *jump* instructions. In debugging the program, the switches are positioned so that the computer will jump when it encounters such an instruction; usually special print-out routines are included in the program, and the *jump* instructions transfer the computer to these routines. When the program has been corrected, the switches are turned off and the computer simply ignores the *jump* instructions.

Use of the Automonitor Program. The automonitor method for dynamic debugging does not require any additions or special considerations in the program itself, as does the check-point method. Instead the automonitor method requires the use of a special automonitor program which is read into the computer in addition to the program to be debugged. The automonitor program is general and once written can be used for every program to be debugged. Occasionally the computer has the properties of the automatic automonitor program already wired in, so that the automonitor process can be carried out simply by throwing a switch on the computer console. An automonitoring program makes the computer *print out—while each instruction of the program being debugged is being executed—the instruction itself, the contents of each of the operand addresses, and the computed result.* For example, suppose

the program to be debugged is as follows:

Address	Instruction				
δ_0	Operation	α_0	β_0	γ_0	δ_1
δ_1	Operation	α_1	β_1	γ_1	δ_2
δ_2	Operation	α_2	β_2	γ_2	δ_3

Then the automonitor program would have the computer print out the following:

Address	Instruction	1st operand	2d operand	Result
δ_0	(δ_0)	(α_0)	(β_0)	(γ_0)
δ_1	(δ_1)	(α_1)	(β_1)	(γ_1)
δ_2	(δ_2)	(α_2)	(β_2)	(γ_2)

In other words, the automonitor program gives a *complete record* for the program being debugged of just which instructions are being done by the computer and what the results were. With this information it becomes easy to spot any error in the program being debugged. Note that the automonitor program is a program that operates on another program—more about this point will be said in the succeeding sections.

Of course certain instructions of the program being debugged are handled somewhat differently. For example, in a *compare* instruction, (γ) need not be printed, but just δ, (δ), (α), (β), for the address of the next instruction tells which way the comparison went. Common sense clearly dictates what an automonitor program should print out.

An automonitor program for a three-address computer would be similar to that just described for a four-address system. For a two- or one-address system the contents of the accumulator or other registers become of importance, and hence the contents of the accumulator is printed out. For instance, in a one-address system, the print-out would look like this:

δ_0 (δ_0) (α_0) (acc)
δ_1 (δ_1) (α_1) (acc)
δ_2 (δ_2) (α_2) (acc)

The Automonitor Program Itself. In order to grasp more fully the general concept behind an automonitor program, let us examine in a cursory way how such an automonitor program itself might be written for a one-address system. The flow chart is given in Fig. 5-1. This automonitor program is in one part of the memory, and the program to be debugged is in another part of the memory. The automonitor program

must be given the address of the first instruction of the program to be debugged. Then the automonitor program prints out the desired record and also executes the instructions of the program to be debugged. First the automonitor program prints the address of this first instruction, δ_0; then it copies (δ_0) into a special temporary located in the automonitor program's part of the memory. This instruction is printed, and (α_0) is printed. Next the automonitor program directs the computer to execute the first instruction *in* its location *within* the automonitor program. Then (acc) is printed, and the automonitor program goes on to the next instruction, etc. There are a few important details that must be attended to. When the program to be debugged is being executed, the contents of the accumulator from instruction to instruction is, of course, important. But when the automonitor program executes one instruction at a time and inserts print routines, etc., between these instructions, the contents of the accumulator becomes lost. In order to correct this situation, the contents of the accumulator (and other registers) must be recorded after the execution of each instruction of the program being debugged. Then, before the next such instruction is to be executed, the registers are properly preset.† In this way the two programs are properly interleaved.

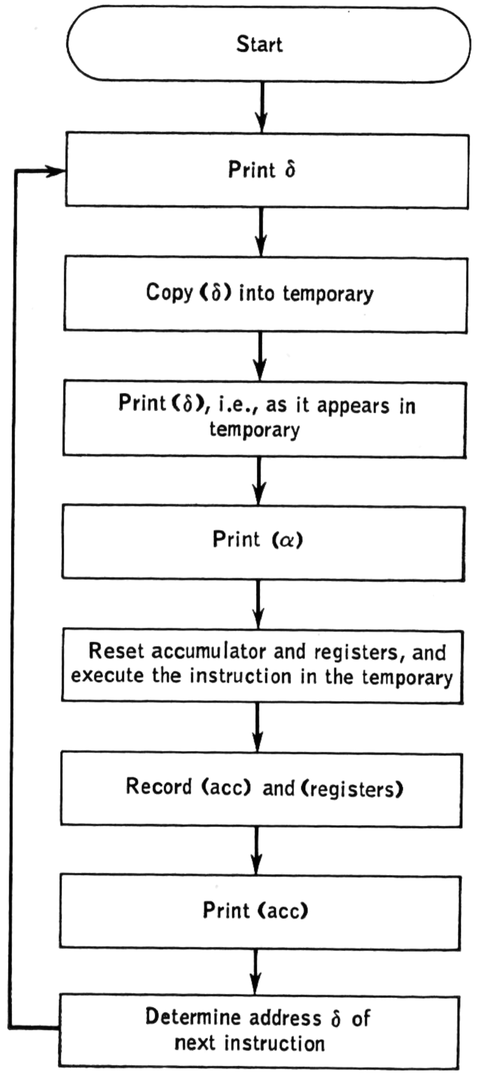


FIG. 5-1. Flow chart of automonitor program.

Post-mortem Program Debugging. Consider next post-mortem program debugging. The first method is to run the program through and observe the results. From a knowledge of the program and the erroneous results an error can often be located. For example, if the magnitude of the

† Also, provision is made in the automonitor code so that *jump* instructions are not performed directly but rather operate on the automonitor δ-address reading instruction.

results is off, the scaling is checked for error, etc. The second method is to print out the entire memory of the computer after the program has been run. If the program got into a loop with no exit, then the tallies can be checked on this memory print-out. All instructions that were to be modified by the program can be checked to see whether or not they were correctly modified, etc. A third method is to have another program that examines the program to be debugged after it has run. This program compares the program to be debugged with itself before and after it has been run and tells the programmer where they differ. In this way the programmer can determine if any instructions were inadvertently destroyed, if all differences that should appear actually do appear, etc.

Of course the programmer uses combinations of the techniques that are at his disposal. What we have briefly outlined here is in no way complete but rather indicates some of the directions from which aids to programming may arise.

5-3. Mathematical Interpretive Routines

Automatic Programming. There are two general kinds of *automatic programming,* called *interpretive* routines and *compiling* routines. The purpose of automatic-programming techniques is to reduce the number of instructions actually written by the coder. The method for accomplishing this is previously to place in the computer a library of subroutines that can be used by every coder, so that these routines will not have to be rewritten each time they are needed. A set, or library, of particular subroutines, together with a method for using them, is called an automatic program. The difference in the two kinds of automatic programs lies in their use: in the interpretive program the subroutines are to be used by the coder's program while it is running on the computer, whereas in the compiling program the subroutines are used by the coder's program only at the time the coder's program is read into the computer.

We have arbitrarily classified interpretive routines into three types: (1) *mathematical,* (2) *simulational,* and (3) *data handling.* The subroutines of the mathematical type of interpretive routines compute values of functions, those of the simulational type are usually designed to make one computer look like some other kind of computer, and those of the data-handling type assist in organizing large amounts of data and in data searching.

Use of Mathematical Interpretive Routines. Let us first consider an example of a mathematical interpretive routine. The purpose of such a routine is to enable a coder to form such functions as $\sin x$, $\cos x$, e^x, $x!$, $\sqrt[n]{x}$, etc., merely by writing a couple of instructions. Of course these functions would appear as subroutines already in the computer, as part of the interpretive routine. However, the interpretive routine must also

contain a section that can interpret the instructions that the programmer writes. When a function subroutine is to be used, the following information must be recorded: (1) the address that contains the value of the independent variable x; (2) the address into which the result $f(x)$ is to be put; (3) the address in the program from which the jump to the subroutine was made, so that after the subroutine has been completed the computer will know where to *jump back* to the original program; and (4) the address to which to jump in order to go to the subroutine itself. Consider a two-address system: two words are needed to go to a subroutine (since four addresses are required). We could arrange the words as in Fig. 5-2. The first word is a *return jump* instruction; the second

Operation	α	β
Return jump	Standard address	Address of 1st instruction of function subroutine
. . . . .	Address of x	Address of $f(x)$

FIG. 5-2. Instruction and address-data word for interpretive routine.

word is not an instruction but is just to tell the subroutine where x is and where $f(x)$ is to be put. Recall that the *return jump* instruction jumps to β, and in addition the instruction memorizes as the contents of α its own address. Hence the return jump takes care of items 3 and 4 above and the second word of items 1 and 2. Thus the programmer, each time he wants to use a subroutine listed in the interpreter library to evaluate a function, need write only the following "instructions": (*a*) a *return jump* instruction, where α is a standard address (standard, that is, for this particular interpreter) and where β is looked up in a book of instructions for the interpreter which lists the proper β for each function; (*b*) the word following the *return jump,* as described above. This has illustrated the procedure for using a mathematical interpreter.

The Interpretive Routine Itself. Now let us consider the interpreter program itself. Besides consisting of the function subroutines, as we observed above, it must also consist of a part that interprets the two words that the programmer writes. This part looks at the standard address to find out where the address of the *return jump* is. Knowing that the next word gives the locations of x and where $f(x)$ should be put, this part of the code then relates that information to the appropriate subroutine and transfers control to this subroutine. This part can be thought of as a subroutine also; we shall call it subroutine Q. The flow diagram might appear as in Fig. 5-3, where, to be specific, suppose that

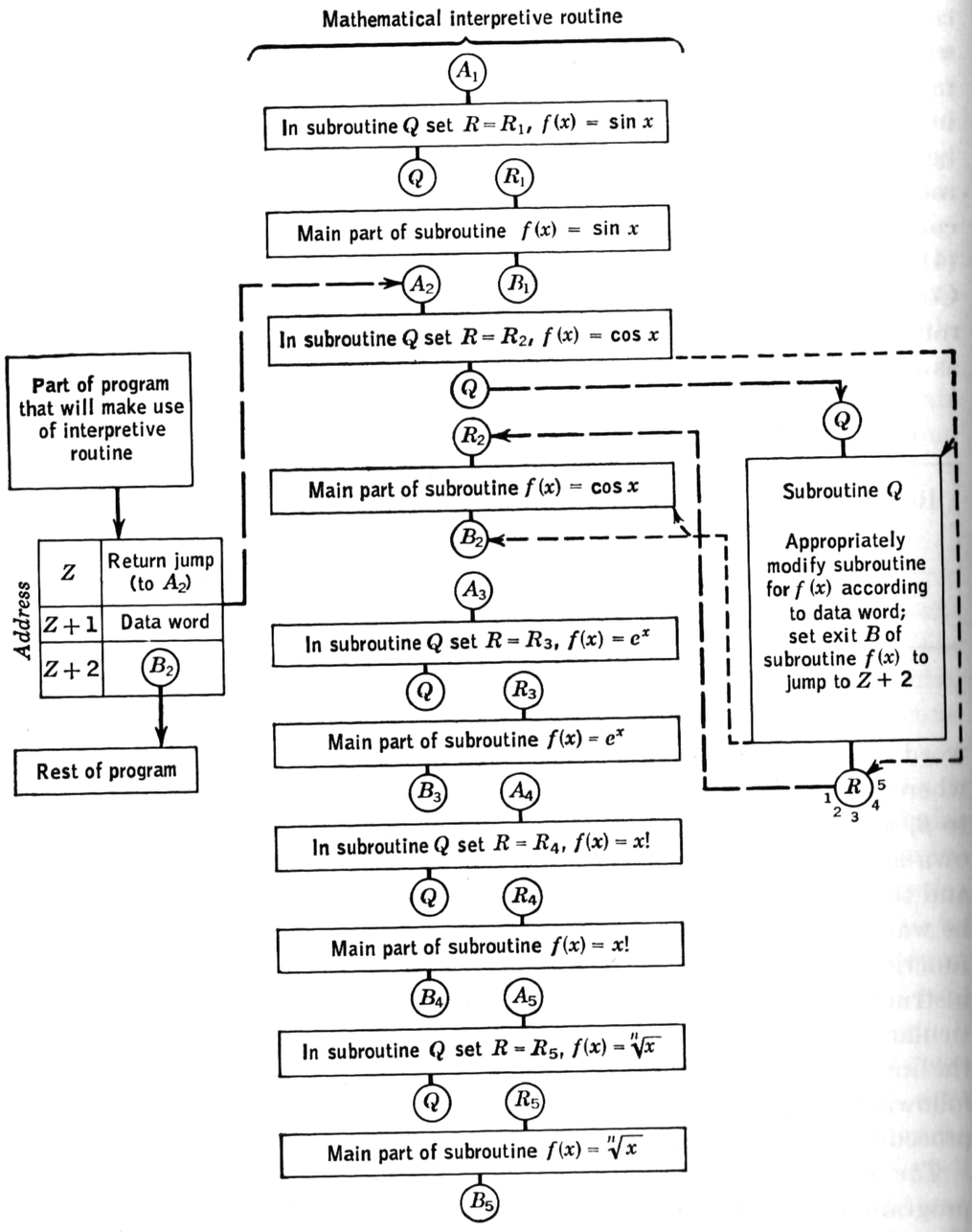


FIG. 5-3. Flow chart of interpretive routine.

it is desired to use a cos x subroutine. The long dashes indicate the path of the computer through the program and subroutines; the short dashes, the operating of one subroutine on another subroutine. We let Z be the address of the *return jump* instruction; then $Z + 1$ is the address of the data word, and the program should resume at address $Z + 2$.

As an example of the type of things that occur in the subroutine Q, consider how it inserts the value of x into the subroutine for $f(x)$. Sub-

routine $f(x)$ has a certain cell reserved for the value of x; call this cell T_x for "temporary for x"; within subroutine Q is an instruction that transfers the value of x to T_x from the address given by the data word of the program. Suppose that the instruction were "Transfer (α) into β." Then Q must have a previous instruction that puts into the α address of the transfer instruction the address of x as given by the data word, i.e., an *extract* instruction that takes the α address of the contents of $Z + 1$ and puts it into the α position of the *transfer* instruction. But this *extract* instruction must involve the address $Z + 1$, which is obtained from the standard address associated with the return jump. Hence there must be previous instructions that take Z as given in the standard address, add 1, and insert this into the α part of the *extract* instruction. Next consider the β part of the x-to-T_x *transfer* instruction: this must be some address specifically reserved in the subroutine of $f(x)$ for the value of x. Since subroutine Q is a general subroutine, it must be told this address each time a subroutine for $f(x)$ is to be used. This is done in the part of the subroutine for $f(x)$ denoted in Fig. 5-3 by "In subroutine Q set $R = R_2$, $f(x) = \cos x$." In this part there is an instruction which extracts the address reserved for x into the *transfer* instruction of subroutine Q. This process is valid since the computer will pass through this instruction before getting subroutine Q. Summarizing, there is an instruction in the subroutine for $f(x)$ that sets β of the *transfer* instruction; and there is a series of instructions preceding the *transfer* instruction in subroutine Q that leads to the setting up of the α address of the *transfer* instruction.

Our example of a mathematical interpretive routine has several interesting features: First, one subroutine uses another subroutine—i.e., each subroutine for $f(x)$ uses the subroutine Q. Such a feature may occur in other ways also. For example, in evaluating cos x, suppose that the power series were used:

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

Then the subroutine for cos x will use the subroutine for $a^x/x!$, where $x = 2n$ in this case. In fact, to use the subroutine for $a^x/x!$, the same technique of the return jump and data word is used (why?). Of course the use of subroutines by subroutines can be carried deeper than two or three. Second, note how many instructions are actually needed to perform the simple transfer described above. The moral of this is that coding a simple job can be complicated on a computer. This, indeed, is the reason that it pays to have a separate subroutine Q rather than to repeat the instructions of this subroutine for each $f(x)$ routine. Finally, the main purpose of this mathematical interpretive routine is to make the coding problem easier by allowing only two words to be written when

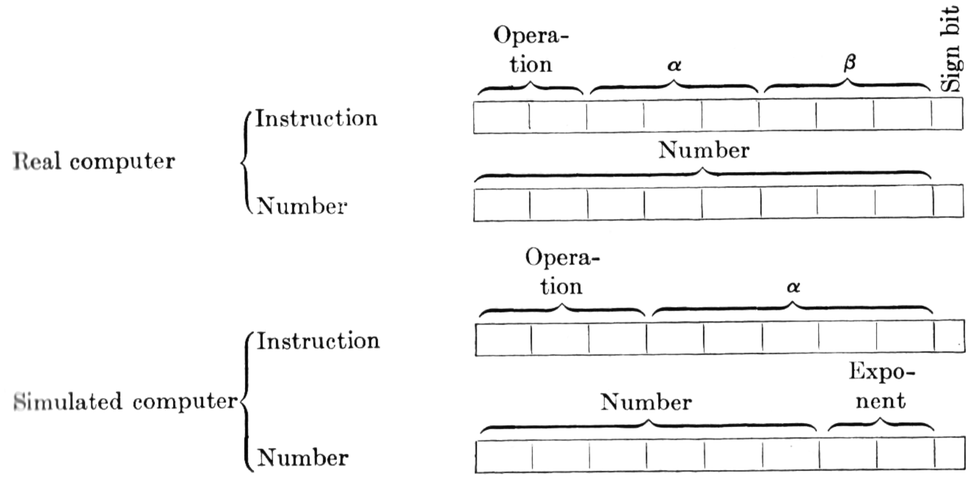
a function is to be evaluated. Naturally the process described here was greatly simplified, since many other complications may arise—telling the $f(x)$ subroutine to what accuracy the function is to be calculated, taking care of the proper scaling, etc.

5-4. Simulational Interpretive Routines

Use of Simulational Interpretive Routines. Next let us consider simulational interpretive routines. The purpose of these routines is to make one computer appear as an entirely different computer. For example, suppose that we had a computer that did *not* have floating operations. By means of the proper routine we can make this computer *appear* to be a computer that has built-in floating-point operations. Or we can make a one-address computer appear to be a two- or three-address computer. Another example might be the making of an ordinary computer into a special-purpose war-gaming computer. Of course the coding for the simulated computer will be quite different from the coding of the actual computer. However, the purpose of the simulational interpretive routine is to enable the programmer to write his code in the language of the simulated computer; he need never know that this is not the actual method of coding for that computer. Obviously, then, the interpretive routine must interpret every instruction of the programmer's simulation-language code. Thus the interpretive routine looks at the first instruction of a simulation-language code, interprets it, then goes to the next instruction of the simulation-language code, interprets it, etc., for each instruction passed through in the simulation-language code. Immediately we see that the interpretive routine must also keep track of what instruction of the simulation-language code to take next; i.e., the interpretive routine must contain a simulated-instruction address counter, a simulated "simulation-language" instruction register, simulated decoder, simulated accumulator, etc. In other words, the simulation routine must perform all the functions that the electronic circuitry would perform if the simulated computer had been real.

The Interpretive Routine Itself. For the sake of illustration let us discuss a simulation interpretive routine that makes a *two*-address *nonfloating* operation computer appear like a *one*-address *floating* operation computer. To be specific, suppose that there are 6 bits in the operation and 9 bits in each of the two addresses of the real computer. Suppose also that in the simulated computer there are to be three octal symbols for the operation and five octal symbols for the address. Of course, since the real machine can address only 2^9 memory cells, even though theoretically 2^{15} memory cells can be addressed by the simulated one-address instruction, only 2^9 memory cells are available. Hence three octal positions in the simulated-instruction format are meaningless. However, one could interpret these as addressing cells on magnetic tape,

etc. In so far as the simulated numbers are concerned, the rightmost two octal positions of a word will represent the exponent, the rest of the positions the value of the number. These formats are then as follows, where each wide box represents an octal digit (i.e., three binary bits):



Let us suppose that the real computer has the two-address operations listed in Sec. 3-1, as well as the following additional instructions:

Code	*Description of operation*
30	Shift (acc) left α positions; put result into β
31	Shift (acc) right α positions; put result into β
20	Extract (α) into β, using (acc) as the mask; i.e., replace those bits of (β) with corresponding bits of (α) in those positions where (acc) has units, and put result into β

Suppose that the one-address instructions to be simulated are these:

Code	*Description of operation*
001	Add (α) to (acc), put result in acc, floating unnormalized
002	Multiply (α) by (acc), put result in acc, floating unnormalized
003	Subtract (α) from (acc), put result in acc, floating unnormalized
004	Divide (acc) by (α), put result in acc, floating unnormalized
005	Conditional jump: if (acc) is negative, take (α) as next instruction
006	Transfer (acc) into α
007	Jump: take (α) as next instruction

Of course the accumulator referred to in these instructions is the simulated accumulator. To be more specific, let address 776 be the simulated-instruction address counter, 775 the simulated instruction register, 774 the simulated accumulator.

The programmer will now code in terms of these simulated instructions only (*never* writing down any of the real instructions). It is up to the interpretive routine to sequence through these simulation-language instructions, interpreting them as it goes along. Obviously the inter-

pretive routine is written in the real machine language. Figure 5-4 is a general flow diagram showing how the interpretive routine sequences the computer through the simulation-language instruction code. The student should study it carefully. We shall first discuss some of the details of box 1 of this flow diagram.

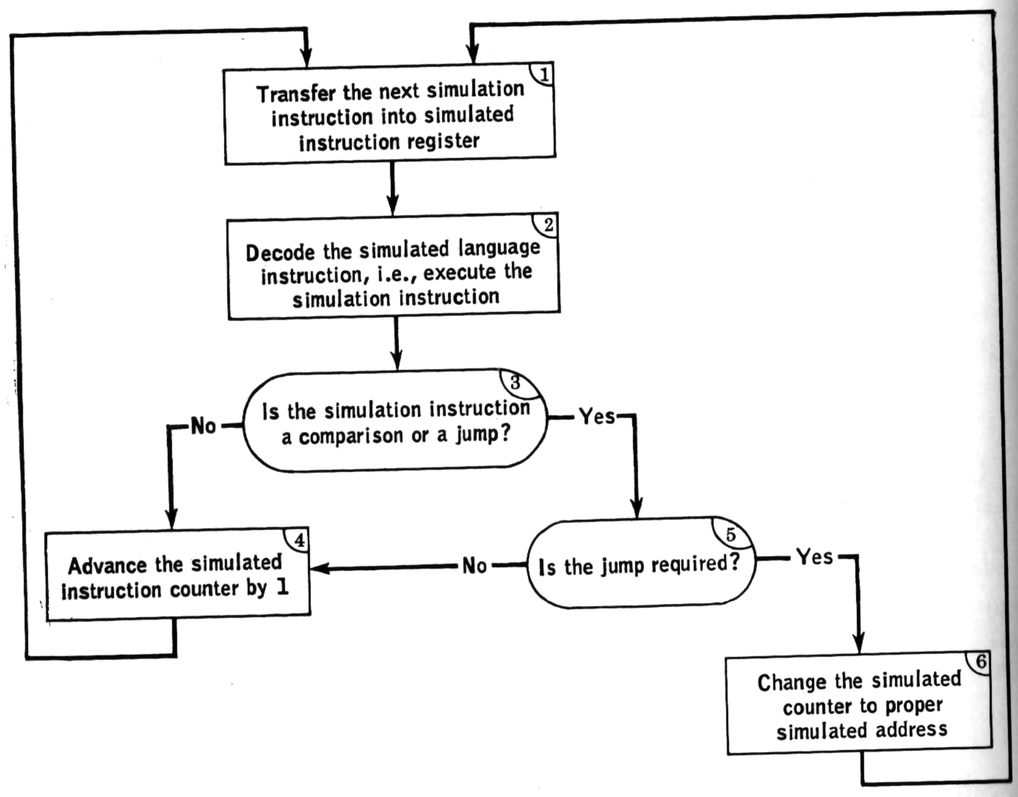


FIG. 5-4. General flow diagram indicating how interpretive routine sequences the computer through the simulation-language code.

The heart of box 1 is a pair of instructions that transfer the proper simulation-language instruction from its location in the code to the simulated-instruction register. Preceding this pair of instructions are instructions that set up this pair to operate on the proper addresses. In symbolic code, the pair might look like this:

Operation	α	β
Add, i.e., 53	Address of a constant of all zeros, i.e., ϕ	Address of proper simulation-language instruction
Transfer, i.e., 52	Simulated-instruction register, i.e., 775	Address of first instruction of box 2

The address of the proper simulation-language instruction is found in the simulated-instruction address counter. Hence the contents of the β part of this word (i.e., the contents of 776) must first be extracted into the β part of our add instructions:

Operation	α	β	Remarks
Add, i.e., 53	ϕ	Address of a constant that has all units in β position and zeros elsewhere	Set up mask
Extract, i.e., 20	776	Address of above *add* instruction	Do extraction

Suppose that we put the first instruction of box 1 into address 600; then altogether for box 1 we would have

Address	Operation	α	β	Remarks
600	53	604	605	Set up mask
601	20	776	602	Do extraction
602	53	604	...	Transfer
603	52	775	606	Transfer
604	00	000	000	Zero constant, i.e., ϕ
605	00	000	777	Mask

Each of the simulation-language operations is performed, of course, by a subroutine contained as part of the interpreter routine. Hence box 2 consists of these subroutines, in addition to a jump table that looks at the simulation-language operation and jumps to the proper subroutine. The *shift* instruction would be used in this procedure. The simulation-language instruction is shifted to the right 17 (*octal*) positions [15 (*decimal*)], putting the operation code in the β position, leaving zeros elsewhere. Then this shifted word is added to the *transfer* instruction that will jump to the proper *jump* instruction of the jump table. In this way the simulated operation code is used to enter the proper subroutine from the jump table. Note that first the *transfer* instruction that jumps to the table must be reset, for it becomes modified each time it is used. Hence, in the first part of box 2, we have the code of Table 5-1. When a type 52, or *transfer*, instruction is used as a jump, the α address does not have much meaning. Hence we always put 777, the address of the real accumulator, for α, to ensure no change due to this part of the *jump* instruction.

Let us examine briefly the subroutine for the simulation-language multiplication, floating unnormalized, as an example of our arithmetic subroutines. The purpose of this subroutine is to multiply (α) by (774), where α is the address found in the contents of 775, the simulated instruction register (see the instruction and number formats for the simulated computer above). The result is to be put into 774, the

TABLE 5-1. CODE FOR BOX 2 OF FIG. 5-4

Address	Operation	α	β	Remarks
606	53	604	610	Put jump constant into accumulator — Reset *jump* instruction
607	52	614	611	Put constant into 614 — Reset *jump* instruction
610	52	777	614	Constant used to reset *jump* instruction
611	53	604	775	Shift simulated-language instruction to right 17 (*octal*) positions in accumulator
612	31	017	777	Shift simulated-language instruction to right 17 (*octal*) positions in accumulator
613	51	614	614	Add shifted instruction to *jump* instruction
614	52	777	614	*Jump* instruction
615	52	777	Add subroutine	Jump table that sends control to proper simulated operation subroutine
616	52	777	Multiply subroutine	Jump table that sends control to proper simulated operation subroutine
617	52	777	Subtract subroutine	Jump table that sends control to proper simulated operation subroutine
620	52	777	Divide subroutine	Jump table that sends control to proper simulated operation subroutine
621	52	777	Conditional jump subroutine	Jump table that sends control to proper simulated operation subroutine
622	52	777	Transfer subroutine	Jump table that sends control to proper simulated operation subroutine
623	52	777	Jump subroutine	Jump table that sends control to proper simulated operation subroutine

simulated accumulator. In order to perform this multiplication, $P7$ through $P24$† of each word are to be multiplied, while $P1$ through $P6$ of each word are extracted into initially empty temporaries, the temporaries are added, and the result extracted into $P1$ through $P6$ of 774. In performing the addition, $P7$ of the sum must be checked for overflow: the exponent of the result cannot be larger than six positions. After performing such a subroutine the computer would go to box 3 of the general flow chart.

5-5. Data-handling Interpretive Routines

Use of Data-handling Routines. The manipulation of large masses of data can become a serious problem to the programmer. Such data must

† See Sec. 1-8.

frequently be interrogated or searched, to find or insert specified items. Also, reasonably efficient organization of the data must be maintained in order that memory space not be unduly wasted. Hence data-handling routines which are designed to assist in such processes must be able to (1) search data, (2) relocate data, (3) insert additional or new data, and (4) omit obsolete or old data. In this section we shall consider two basic organizational concepts for formulating interpretive routines to assist in the programming of such tasks: the first can be visualized as a bookshelf organization, where the data is searched and moved around in a manner akin to the handling of books; the second involves the use of "threaded" data components, or lists where conceptually contiguous data components may physically be spread diffusely throughout the computer's memory, but where each data component gives the address of the next successive component. (We might suggest at this point that, for a fuller understanding and appreciation of the utilization of data-handling techniques, the student refer to the discussions on searching, sorting, and ordering of Chap. 11.)

The Conventional Search. Data is basically organized in terms of lists, such as inventory lists, billing lists, bibliographical lists, subject word lists, and so forth. Suppose first that our collections of lists are recorded on magnetic tapes, and recall from Chap. 3 that magnetic-tape read-in, read-out, and tape-moving instructions usually specify the number of blocks to be read in, read out, or moved. Hence only multiples of blocks can be handled. Therefore, in recording a list on a section of magnetic tape the list is usually padded at the end with words of all zeros so that the entire list will make up an integral number of blocks. The first word of the first block of the list is often used as a labeling word: it contains information about the contents of the list, and, most important for our discussion, it tells how many blocks long the list is. The magnetic tape will then consist of a sequence of such lists, each labeled with the number of blocks in its length as the first word of the list.

There are two advantages to labeling the lists: (1) The computer will at first observe the label word of the first list. If this list is not the one desired, the computer can then move the magnetic tape by the number of blocks indicated in the label, past this list to the next list; the label of the next list will be observed, and so forth. In this way the computer may jump over parts of the magnetic tape not of interest during the particular search. (2) When a list is of interest, then each word of the list must be examined; this is usually accomplished by some kind of iteration, or loop. The tally of this loop can be set according to the length of the list to be examined, as given in the label word.

For example, in the second method for searching given in Chap. 11, each characteristic is followed by a list of its associated items. Hence the first, or label, word of each item list will denote the characteristic

and the length of the list, say in the α and β positions for a two-address system. Then the computer will look for the first of the given (ordered) characteristics by jumping over lists until the appropriate list label is obtained.

A similar process can assist the searching of lists recorded in the high-speed memory of a computer. Again the lists are labeled, and again the label contains information about both the contents of the list and how

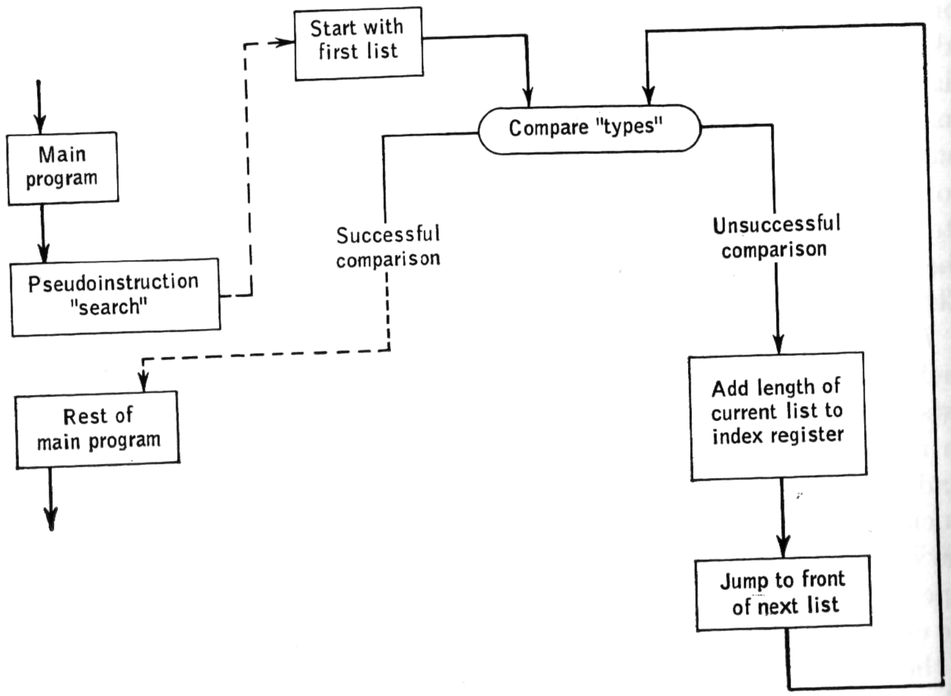


FIG. 5-5. Illustration of use of interpreter routine for the pseudoinstruction "Search."

many words there are in the list. The computer can then observe the label of a list, and if it is not the desired list, the address of the label word of the next successive list can be obtained by, say, adding the length of the current list to an index register. The advantages of the label are the same as discussed for the magnetic-tape case.

If a convention is fixed for the format of the label word of a data list, then an automatic interpreter routine can be written that will be able to interpret a pseudoinstruction *search*. For example, suppose the "label" is to consist of two words at the front of the list, the first giving the type of items in the list and the second giving the length of the list. A *search* instruction would contain the type of the list being searched for: the interpretive routine would compare the "type" specified in the instruction with the "type" of a list; if the "type" did not match successfully, then the computer would add the length of the present list to an index register and jump to the next list; if the "type" of the list were the

desired type, then the interpretive loop would be exited and the computer would proceed with the next instruction in the main program (see Fig. 5-5).

Of course the searching process might be more complicated. Two such additional complications will now be discussed. First, after a desired list is found, it may next be desired to find a certain sublist of this list. Another *search* instruction would then be in order, but care must be

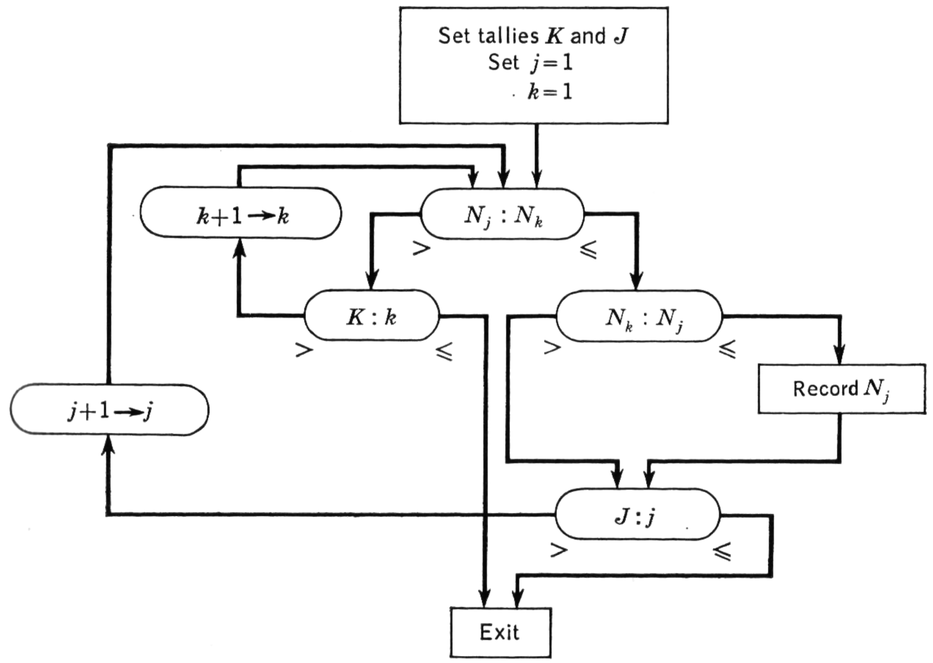


FIG. 5-6. Flow chart to record common numbers on two ordered lists.

taken in the definition and interpretation of such a sublist *search* pseudoinstruction that the sublists of the list are searched. Similarly the search of the original list should skip consideration of sublists. A simple method for avoiding confusion between list and sublist searching is to include in the label of a list or sublist an identifying character that can be sensed by the pseudoinstruction interpreter routine.

The second complication to be considered here is that finding the desired list usually does not end the search. Frequently a next step (see Chap. 11) involves the basic operation of the comparison of two ordered lists for common numbers. Hence we can have another pseudoinstruction for performing this task. Suppose that we have two ordered lists, with K numbers in the first list, indexed by k ($k = 1, 2, \ldots, K$), and J numbers in the second list, indexed by j ($j = 1, 2, \ldots, J$). A detailed flow diagram of a code to record every number that is common to both lists is shown in Fig. 5-6. If the number in the j list is

greater than the number being compared with it in the k list, then the code moves along the k list until a common number is obtained, or until a number in the k list is reached that is greater than the one in the j list. Then the code moves along the j list in a similar way, until the k number is equaled or exceeded. This process depends basically on the fact that the j and k lists are ordered.

Close Up and Make Space. The relocation of data lists, the insertion of new lists, and the omission or erasing of old lists are all important parts of data handling. If the lists are recorded in contiguous parts of the memory or tape, then the handling of the data must be accomplished so that the components of a list remain in adjacent memory addresses and so that spaces are not inadvertently formed between lists. Two pseudoinstructions that are useful under these circumstances are "Close up space" and "Make space." For example, if a new data list is to be inserted between two lists already stored in the memory, these already stored lists must be separated in order that a number of words equal to the length of the new list be made available between them; here the *make-space* instruction would be utilized. This of course means that all the lists after the new list will have to be "moved over" to accommodate the new list. Similarly if an old list is to be erased or omitted, the *close-up-space* instruction will be used to move over the rest of the remaining lists to fill the space vacated by the omitted list. The relocation of a list utilizes both these pseudoinstructions, for first space must be made in the new location, and after the move, the vacated space must be closed up. These processes can be visualized as being analogous to making space for a new book on a bookshelf, moving the books together when a book is taken out, and rearranging books on the bookshelf (see Fig. 5-7).

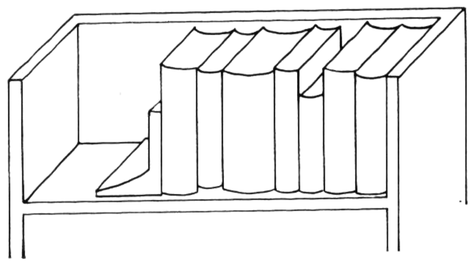


FIG. 5-7. Analogy of book shelf with data-manipulation processes.

The pseudoinstructions must of course utilize the address of the first word of the list to be erased, or the first address of the location where the list is to be inserted. The label word of the list itself tells how many addresses are to be involved. The interpretive routine must first determine these two items of information before performing the move. The interpretive routine, however, must know when the moving job has been completed. This is accomplished by using an *end-of-data* word, analogous to the book end of our bookshelf analogy. To close up space, each data list is moved over until the end-of-data word is sensed; then this word also is moved over, and the process is terminated. To make space, the interpretive routine searches for the end-of-data word, moves it over the appropriate number of addresses, and then proceeds to move the data lists over until the new space appears in the proper location.

Threaded Lists. The contiguous-component list structure which we have been discussing has two *dis*advantages. First, in order to make or close up space, all the lists after the space must be moved. Second, there was no means for dealing with the components or elements of the list. The concept of threaded-component list structure does not have these disadvantages. In its simplest form the idea is that each word which is a member of a list has as part of its contents the address of the next word in the list, in a manner similar to the instructions of the four-address or one-plus-one-address systems. Of course a collection of words can be associated with the list word as a label. In addition to the address of the next word on the list, the list word may include in its contents a "type" designator, which identifies it as being associated with some type "value." Then special instructions can be devised that will look through a list until a list word of a specified type value is found. A particular threaded list will not necessarily have its components in adjacent memory addresses but will generally have its components spread diffusely throughout the memory.

The basic concept, then, is to have the memory entirely organized in terms of lists, including a list of the as yet unused memory locations. The interpretive pseudoinstruction system of the computer would never explicitly deal with addresses, but with lists and type values. Addressing would be accomplished automatically by means of the list words. The exact location of any item in the memory would, in concept, not be explicitly known; but the item could be extracted from the memory by means of list processing.

The use of threaded lists and sublists involves a notation enabling traces to be made through sublists of lists. As an example of such a notation for threaded sublists, let the words of the sublists be composed of three parts. The first part gives the address of the next word on the list under consideration; the second part gives either the first address of a sublist that branches at that point or the "type" value if no branching is to occur; the third part is, say, the character E if the word is the *end word* of a list or sublist, B if a branching point occurs at that word, and C if neither of these two cases occurs. It should be noted that, if the word is an end word, then by convention the first part is to give the address of the first word of that sublist so that the main line can be continued. For instance, Fig. 5-8 illustrates a list and sublists. Here T represents some type-value code. A list or sublist is usually denoted by its first address; thus, for our example, list 1 has two sublists, 2 and 8, where sublist 2 itself has a sublist 3 and sublist 8 has sublist 9, which in turn has sublist 10.

Push-down Lists. The push-down list can be conceived as a non-branching threaded list made up by placing each new element "on top" of the list in the same memory location, each time "pushing down" the other elements; when an element is removed from the list, it is always

Address	Part 1	Part 2	Part 3
1	2	*T*	*C*
2	7	3	*B*
3	5	4	*B*
4	3	*T*	*E*
5	6	*T*	*C*
6	2	*T*	*E*
7	8	*T*	*C*
8	15	9	*B*
9	14	10	*B*
10	12	11	*B*
11	10	*T*	*E*
12	13	*T*	*C*
13	9	*T*	*E*
14	8	*T*	*E*
15	1	*T*	*E*

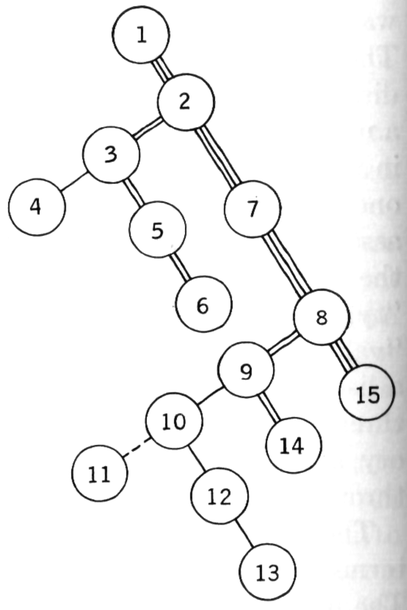


FIG. 5-8. Illustration of a list and its structure. The three parallel lines represent one list, the two parallel lines its sublists, the single lines the sublists of the sublists, etc.

Address	List elements
1	*A*
2	
3	
4	

Address	List elements
1	*B*
2	*A*
3	
4	

Address	List elements
1	*C*
2	*B*
3	*A*
4	

(*a*) "pushing down"

Address	List elements
1	*B*
2	*A*
3	
4	

Address	List elements
1	*A*
2	
3	
4	

(*b*) "popping up"

FIG. 5-9. Illustration of push-down-list processes.

taken off the top by removing it from this same memory location and the remaining elements are then made to "pop up" (see Fig. 5-9). Of course it is not desirable to move the contents of words literally from one memory location to another. Rather it is more efficient effectively to change the addresses of the memory locations by means of a list index register that records the address of the next word on the list. Then pushing down and popping up is analogous to moving backward and forward along the list chain, where the address of the index register becomes the effective address of the top element on the list. To pop up, the address of the next word in the list is inserted in the list index register; to push

down, the address of the new list word is inserted into the register. New words for the list must of course be appropriately formed as described above.

An example of the use of a push-down list is for the unused memory locations in a threaded-list organized memory. When required, unused memory locations can be obtained from this list; vacated memory locations can be put on the top of the list. Also subroutines that use *themselves* as subroutines can utilize push-down lists to record the independent parameters at any level of use (why?). Threaded lists and push-down lists are closely associated with each other. For example, in tracing through a threaded list, the stage of the sublist trace can be recorded in a push-down list of first words of successively nested sublists.

5-6. Memory Space, Speed of Computation, and Automatic Programming

Sharing of the High-speed Memory. As we have seen above, a computer can have a high-speed memory (often magnetic cores), a medium-speed memory (magnetic drums), and finally, a slow-speed memory (magnetic tapes). As one might guess from our previous discussions, the high-speed memory may often be filled when an interpretive routine is stored in addition to a program. Hence careful attention must be paid to the location of the interpretive routine so that it will not interfere with memory space required for the program. It may often happen that the interpretive routine has to be stored on the drum. This may slow up the rate of computation because, as will be recalled, in order to use a word from the drum one must wait for the drum to turn to the proper address before a word can be read out.

This problem may be approached as follows: At the time a particular subroutine of the interpretive routine is to be used, this subroutine is transferred out of the drum into the high-speed memory, where the computations take place. In this way subroutines of the interpretive routines will be performed in the high-speed memory even though they are usually stored in the drum. However, this means that subroutines must be continually transferred out of the drum and into the high-speed memory and back into the drum, which takes precious time. Also, we have added a new requirement for the interpretive routine to fulfill. Now it must not only interpret the simulated instructions but must also *transfer* from the drum to the high-speed memory the subroutines of the interpretive routine as they are needed.

Complications. This problem becomes quite complicated when one is desirous of putting some flexibility into the procedure; for, in general, when transferring a subroutine from the drum to the high-speed memory it is best to be able to put the subroutine in that part of the high-speed memory which may not be in use at the moment. Hence it must be determined where in the high-speed memory to transfer the subroutine.

This means that a subroutine may be located in different places when used at different times. Obviously the assignment of addresses often depends upon the exact location of the instructions being performed. Hence the interpretive routine must be able to translate this subroutine and fix up the addresses as required, depending upon the location of the subroutine. We shall discuss this problem in the next section in more detail.

The complexity of an interpretive routine can be increased still further. Observe, for example, that a function subroutine may be used repeatedly in the course of computation and hence it might be desired to leave it in the high-speed memory after it has been transferred there. Other subroutines that are transferred to the high-speed memory may have to be used only once in that program and hence after their use need not take up room in the high-speed memory. Therefore a distinction must be made as to subroutines that should be transferred temporarily. Further, during the computation for a problem, one subroutine may use another subroutine. Suppose that both these subroutines have already been transferred to the high-speed memory. Somehow the interpretive routine should remember not only that these subroutines have already been transferred to the high-speed memory so that the transfer will not have to be repeated, but also where they have been put. The solutions to these problems are too complex to be treated in this textbook. Our purpose has been to give the student an appreciation of the complexities that can develop in planning and writing interpretive routines.

Advantages of the Compiler Routine. Until now we have been considering the use of subroutines as the interpretation of simulation instructions *during* the running or computation of a program. An alternative method is initially to replace the simulation instructions with the proper subroutines before the program is run. That is, the program is written in terms of simulation instructions, but as the program is read into the computer, these simulation instructions are automatically replaced by the proper subroutines—the subroutines then becoming an integral part of the program. Of course, just as the interpretive routine handled the subroutines in our above discussions, so in this new case an automatic routine called a *compiler routine* performs the task of initially inserting the proper routines into the program. Examples of compiler routines will be described in the next section; in this section we wish only to discuss why such routines are used.

The first advantage of a compiler routine over the interpretive routine is that, since in the compiler routine the subroutines are initially amalgamated into the program before the problem is run, the necessity for transferring subroutines from drum to high-speed memory is eliminated. Hence the problem of *time* lost in transferring subroutines during computation no longer appears, because the subroutines will already have been

made part of the main program. Another important advantage of the compiler routine over the interpreter routine is that, since the compiler routine is only used initially, it need not be stored in the computer when the compiled program is running. The interpreter routine must always remain in the computer memory since it operates during the running of the program. Hence, when a compiled routine is run, more *space* is available for the main program.

On the other hand, as can immediately be seen, after all the required subroutines have been incorporated into the main program, this program can be exceedingly long. Some of the *space* advantage of a compiler routine is thereby lost. The usual method for handling such long programs is to divide them up into sections, each of which can fit entirely into the high-speed memory. Then one section at a time is put into the high-speed memory and computed. Meanwhile the other sections are stored in the drum and on the magnetic tapes. Of course a relatively great amount of time is spent in changing sections from drum and magnetic tape to high-speed memory, and hence some of the timesaving advantages of the compiler routine are lost. The relative advantages of a compiler routine over an interpretive routine, if any, depend on the particular characteristics of the routines in question as well as on those of the program to which the routines may be applied. Further, compiler routines can become very complicated, for the compiler routine must break the program into the proper sections and assign the sections to parts of the drum or magnetic tapes. It must also form instructions additional to the main routine which will change sections at the proper time. Another problem arises when a subroutine appears in a program many times. For example, suppose that the program uses the sine routine often. It would be wasting space to compile the sin x routine each time it is required, having duplicates of this sin x routine incorporated into the program. It would be much wiser to have only one sine routine in the memory and to use it in the manner of a usual subroutine. If the program is compiled as it is read in, the compiling routine must sense that the sin x subroutine will be used many times and so must arrange to have it as a subroutine. Naturally, if we have too many subroutines, so many that we have to put some in the drum, then we are in the dilemma described for the interpretive routine.

In the previous sections we have discussed interpretive routines that enable a program to utilize subroutines while it is being computed. We have seen that owing to lack of high-speed-memory space this can be a slow process. The alternative, namely, that of compiling a program as it is read in, is considered in the following section. We must always keep in mind that the major object in using either interpretive or compiling routines is to reduce the amount of coding that the programmer must do.

5-7. Translator and Address-assigner Compiling Routines

Kinds of Compiler Routines. The purpose of a compiling routine is to reduce the amount of necessary coding by allowing the programmer to write simulated-language instructions which are interpreted by the compiling routine. The difference between interpretive and compiling routines is that a compiling routine can be erased from the high-speed memory after it has completed its task. Evidently a compiling routine can compile mathematical subroutines as well as interpret and compile simulation subroutines of the kind described in Secs. 5-3 to 5-5. However, there are other uses of compiling routines that lend themselves more to the technique, and this section and the one following are devoted to these. We shall consider four such uses: (1) as a *translator*, (2) as an *address assigner*, (3) as a *decoder of algebraic symbols*, and (4) as a *decoder of recursion, or loop, symbols.*

Translator. Suppose that we are considering a decimal-coded binary computer for which we have written a symbolic code. Ordinarily specific addresses would have to be assigned before the symbolic code were put into the computer. However, it is possible to construct a compiling routine that enables the symbolic code itself to be read into the computer, the addresses being assigned automatically by the compiler. Such a compiler is called a translator, since it translates from symbolic language into computer language. In these discussions the student should keep in mind that two codes are under consideration: the compiler routine itself and some program upon which the compiler operates.

To be specific, let us consider a one-address decimal-coded binary instruction format having two characters for the operation and three for the address. Suppose, for simplicity in illustration, that the program is to add two numbers, 8,818 and 7,080. The symbolic code for this might be

Address	Instruction		Remarks
	Operation	α	
CLR	MT	ZRO	Clear accumulator
A1C	AD	1ST	Add 1st number to accumulator
A2C	AD	2ND	Add 2d number to accumulator
TRR	TR	SUM	Transfer result into sum temporary
1ST	08	818	1st number
2ND	07	080	2d number
ZRO	00	000	0 constant
SUM	...	...	Result: sum temporary

If the compiler routine is to assign addresses to these instructions and numbers, then—in addition to the instructions themselves, in symbolic form—the *symbolic address* corresponding to each instruction, constant, or temporary must also be read into the computer. We might make the convention that the symbolic address will be read in immediately preceding its associated symbolic contents. For example, for our code we would initially read the following into the computer:

	CLR	Instructions
MT	ZRO	
	A1C	
AD	1ST	
	A2C	
AD	2ND	
	TRR	
TR	SUM	
	1ST	Constants and temporaries
08	818	
	2ND	
07	080	
	ZRO	
00	000	
	SUM	
00	000	

As yet there is no way for the compiler to distinguish between instructions and constants or temporaries. If the convention were made that only instructions will appear first and that all the constants and temporaries are to be put at the end, and if, in addition, the number of instructions in the program were to be given to the compiler, then the compiler would be able to distinguish the instructions from the constants and temporaries. Of course the desired first address of the program must also be given to the compiler. Hence one word would precede the symbolic code, with the convention that the number of instructions appears in the operation positions and the first address appears in the α positions. For our example this first word would be

04 600

if we desired the code to begin at address 600. Of course the symbolic code with its first word must be read into a standard position in the computer memory so that the compiler routine will know where to work.

Suppose that the convention is made that the symbolic code is to be read into the memory starting with the first word in address 001.

Writing the Translator Compiler Routine. It now remains to discuss how to form a compiler routine that will assign specific addresses to a code when all the above-described conventions are adhered to. A general flow diagram for such a compiler appears in Fig. 5-10.

Consider box 1 of Fig. 5-10: First the real addresses would be assigned to the symbolic addresses of the instructions. By our conventions it is known that these symbolic addresses are found in every other word of the symbolic code, starting with the second word. The first symbolic address is assigned to the address found in the α position of the first word of the code. Next the assignment of real addresses for the constants and temporaries is made. This is done by looking at the contents of each instruction: if the symbolic address (in its α position) is not that of another instruction, then it must be that of a constant or temporary. Such addresses are then assigned in sequence. (Note that we cannot have the compiler assign addresses to the constants and temporaries in the same manner as it did for the instructions since we have not informed the compiler of how many constants and temporaries there are.) A detailed flow diagram of box 1 appears in Fig. 5-11.

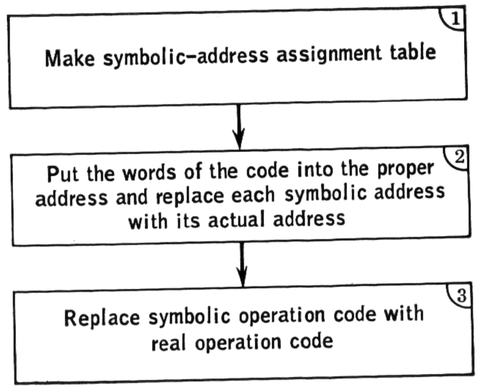


FIG. 5-10. General flow chart of a compiler routine.

Now that a table has been computed assigning real addresses for the symbolic addresses, the instructions and constants can be transferred into these locations, with the α positions of the instructions changed from the symbolic to the real address. Our convention of preceding each instruction and constant by its symbolic address again plays an important role. When the code is compiled into its real position, the instructions and constants will appear in the usual consecutive sequence.

Finally, we must replace the symbolic operation code with the real operation code. For this the compiler routine must be given a table relating these operation codes.

Several further details must be observed. There are some instructions in which *no* address appears in the α positions, e.g., *shift* instructions where the number of positions to be shifted appears instead. Care must be taken that this is not interpreted as a symbolic address to be assigned a real corresponding address. Hence the instruction itself must be examined by the compiler to ensure that it is not one of these exceptional cases. Also, some constants may involve a symbolic address. Care must be taken to recognize these and replace them with the proper real address.

Summarizing, we can draw the following conclusions from our example: First, the writing of a compiling routine can be "tricky" if we are to be certain that all possibilities are taken care of. Second, in order to use a compiler routine, the programmer must adhere carefully to all conventions required by the routine.

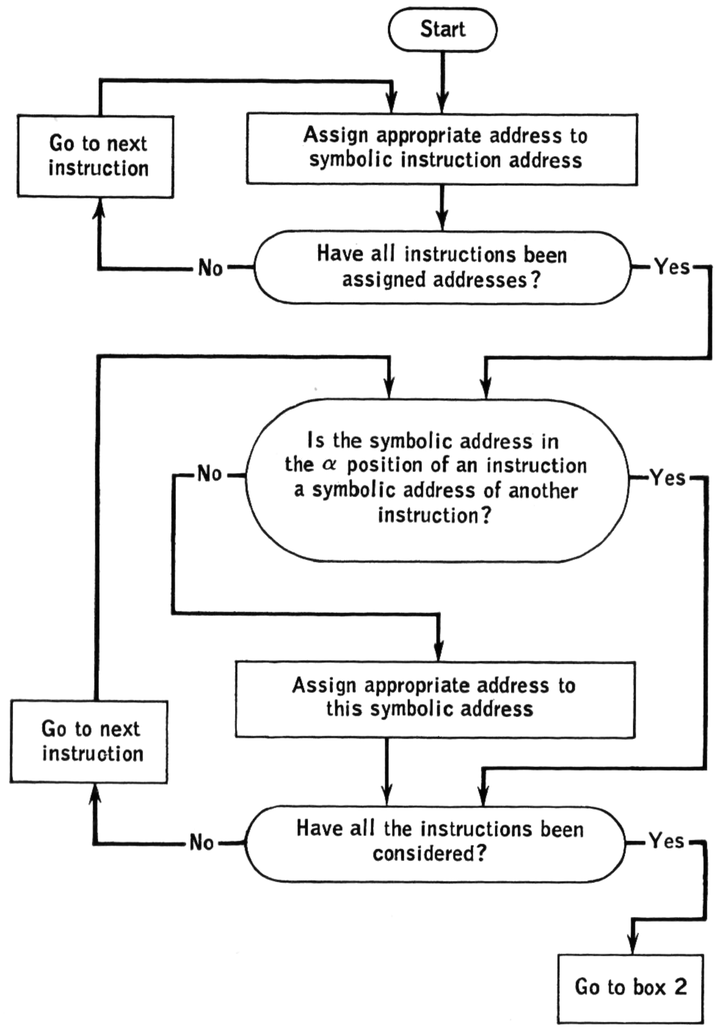


FIG. 5-11. Detailed flow chart of box 1 of Fig. 5-10.

Address Assigner. Suppose that it is desired to write many subroutines and it is not known beforehand where they are to be located in the memory. One procedure might be to write each subroutine as if it were to be located with its first instruction in some standard address,

say 003, and then later to use a compiler to assign new addresses to each subroutine when the final decision is made on where it shall be located. (Recall that a similar requirement arose in connection with the transferring of subroutines from the drum to various locations in the high-speed memory.) To perform this task, the compiler routine merely has to add *to each address* appearing in the code the difference between the desired address of the first instruction and 003. The only problem is to determine which are the addresses. For some instructions such as *shift* instructions do not contain addresses in all the address positions, and these positions should not be changed. On the other hand some constants may contain addresses, and these must be recognized and changed. One procedure is to require the programmer to list all the instructions first, then any constants that contain an address, finally any other constants. In addition, the programmer is required to insert words in front of the program that list (1) the desired address of the first instruction, (2) the total number of words in the code, (3) the total number of instructions in the code, and (4) the total number of instructions that contain addresses. Suppose that we were considering a two-address system; we could then put these numbers, respectively, into the α and β positions of the word in 001 and the α and β positions of the word in 002.

The compiler routine will first determine which instructions contain addresses and will add to these addresses appropriately. Next the program will look in the α and β positions of the constants that have addresses—recognizing these addresses as numbers between 003 and 003 + the total number of words in the code—and will add to these addresses appropriately. Finally the compiler routine will transfer the code from the standard position to the required position.

5-8. Algebraic Compiling Routines

Decoder of Algebraic Symbols. The purpose of this compiling routine is to enable the programmer merely to type into the computer an algebraic formula, the compiling routine automatically writing the detailed machine-language code. Of course the programmer must also tell the compiling routine in what addresses the values of the independent variables will be found and into which address the result is to be placed. Let us first consider a function involving only the operations addition and multiplication, such as

$$((((a + b) \cdot c) + d) \cdot e)$$

In order to avoid ambiguity, certain conventions must be observed. A dot is to be used for multiplication; i.e., we must write $a \cdot b$, not ab. Also the parentheses must be placed wherever necessary and never left

understood. For example, $a + b \cdot c$ is never written, but $(a + (b \cdot c))$ instead. There is to be a parenthesis at the beginning and at the end of the function. Also the parentheses are to indicate binary operations only—i.e., operations involving two operands and no more. Thus one would never write $a + b + c$, but $((a + b) + c)$ instead. Actually, as we shall see in the following chapter, such rather artificial conventions are really not necessary, and more familiar conventions can be used equally well. However, for pedogogical purposes, the use of parentheses as described will enable us to concentrate fuller attention on the basic concepts and problems involved in constructing algebraic compiling routines.

The parentheses play an important role in such an algebraic formula—for they tell in what order operations are to be performed. For example, in

$$((((a + b) \cdot c) + d) \cdot e)$$

the order in which the operations must be performed is $a + b = w$; $w \cdot c = x$; $x + d = y$; $y \cdot e = z$, where z is the final result. A study of the proper use of parentheses shows that they are never ambiguous.† To see which operations should be done first, count the parentheses, say from left to right. Start with the first parenthesis, and let each successive left parenthesis increase the count by one more than the previous (left or right) parenthesis and each successive right parenthesis decrease the count by one less than the previous parenthesis.

Consider, for example, the following:

$$\underset{12}{(}(a \cdot \underset{3}{(}b + c\underset{21}{)}) + \underset{2}{(}d \cdot \underset{3}{(}e + f\underset{210}{)}\underset{210}{)}$$

Notice that the right parenthesis of a pair is counted one less than its left-hand counterpart in the count and that the last right-hand parenthesis takes a count of 0, which was of course the purpose for the above numbering rules. Also there is one pair of parentheses for each operation + or · to be performed.

Now we can gain some insight into how a compiling routine might work to code an algebraic function automatically. To evaluate the functions, we start counting the parentheses according to the above rule in a left-to-right scan. When the count decreases by 1, this is an indication that an operation is to be performed. The operation is performed, and the count proceeds. The process ends when the count is 0. For our example $\underset{3}{(}b + c\underset{2}{)}$ is computed first; this set of symbols is then replaced by the

† See S. Kleene, "Introduction to Metamathematics," p. 24, D. Van Nostrand Company, Princeton, N. J., 1952.

result of the operation; i.e., let $s = b + c$, whence we now have

$$((a \cdot s)_{12} + (d \cdot (e + f))_{1})$$

Next compute $(a \cdot s)_{21}$, and let $t = a \cdot s$, forming $(t_1 + (d \cdot (e + f)_{32})_{2})$

Next compute $(e + f)_{32}$, and let $u = e + f$, forming $(t_1 + (d \cdot u)_{21})_{2}$

Next compute $(d \cdot u)_{21}$, and let $v = d \cdot u$, forming $(t_1 + v)_{10}$

Finally compute $(t_1 + v)_{10}$, resulting in the final value w

Writing the Algebraic Compiler. The compiling routine is to write instructions that will evaluate the function in a manner directly analogous to the above illustration. Figure 5-12 is a simplified flow chart of an algebraic compiler. The algebraic function can be considered as a list of symbol codes $S_1, S_2, \ldots, S_n$ which the compiler is to interpret. The compiler must have available a list of the codes for right and left

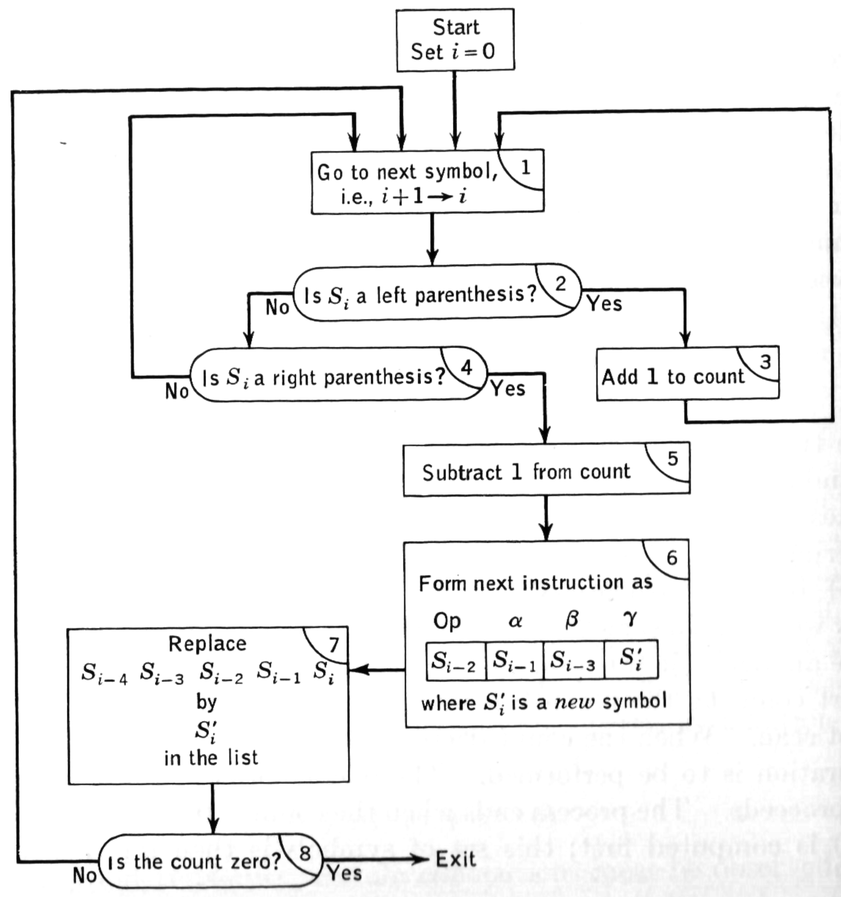


FIG. 5-12. Simplified flow chart of an algebraic compiler routine utilizing the parenthesis convention.

parentheses, arithmetic operations, and letters. As an example utilizing a three-address system, it is readily seen that the flow chart of Fig. 5-12 when applied to our illustration will result in

ADD	c	b	s
MLT	s	a	t
ADD	f	e	u
MLT	u	d	v
ADD	v	t	w

This symbolic code can now have addresses assigned to it, or the address assignment and compiling could have been accomplished all in one step.

Of course we have neglected many important features of an algebraic compiling routine. First observe that the symbols in the final code must represent addresses. Hence, if numerical constants appear in the original formula, these must first be assigned to memory locations. This can be accomplished in an initial processing of the raw input, or in box 1 of the flow chart. We have described algebraic functions utilizing + and · only; the flow chart, Fig. 5-12, will work equally well for subtraction, −, and division, /. However, the flow chart must be extended at box 4 for inclusion of exponentiation, subscript calculation, roots, and so forth, in which case the operation must be more fully analyzed. Further analysis of the operation type would also be necessary if a more conventional notation than that of the parentheses were used, such as interpreting $a + bc$ as $a + (b \cdot c)$. We have illustrated a three-address system; however, the flow chart becomes somewhat more complicated for a one-address system. For example, division is more efficiently coded in a right-to-left scan (see Exercise 5-22). In addition a real compiler should make informed decisions between floating- and fixed-point representation for each constant and partial result. The compiler should utilize the commutative law for addition and multiplication to save on instructions, and so forth.† Note that five groups of memory addresses are involved in this compiling routine:

1. The locations of the compiling routine itself
2. The locations of the words punched from the original symbols
3. The location of the table relating the letters of the function to the addresses that will contain the values of these letters as independent variables
4. The location of the values of the independent variables
5. The location of the program written by the compiling routine

When the compiling routine is writing the function-evaluation program, (1) to (5) are needed; when the resulting program is being run, only (4) and (5) are needed.

† See, for example, R. W. Floyd, An Algorithm for Coding Efficient Arithmetic Operations, *Communs. ACM* vol. 4, no. 1, pp. 42–51, January, 1961.

Recursive Compiler. Consider the problem of constructing a compiler program that will automatically code loops. This problem can become very complicated, depending on the generality with which it is desired to have the program perform. However, to illustrate the concept, let us consider the following oversimplified example: Suppose that we wish to construct a compiler that will evaluate a single function by successive substitutions—in other words, one that will automatically construct a

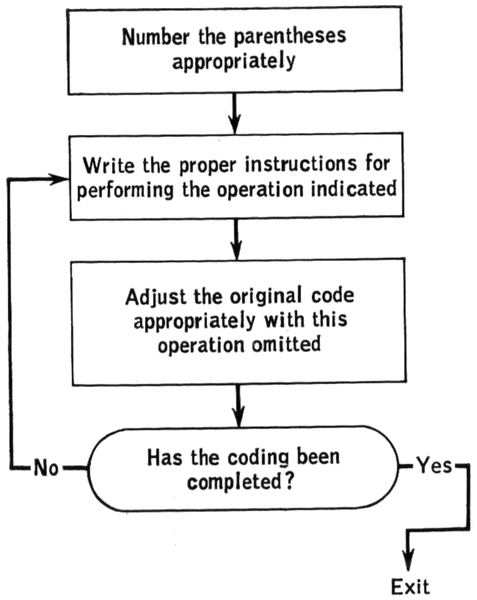


FIG. 5-13. Flow chart for the evaluation of a single function.

code to perform the flow diagram of Fig. 5-13. Here let us assume that the function $f(x)$ is given by a subroutine: then the only information the compiler needs is the number of the subroutine for $f(x)$, the initial value of x, namely x_0, and the accuracy ϵ for which the function is to be evaluated. Of course in this oversimplified case we are tacitly assuming that $f(x)$ has certain properties (why?).

One very simple way of constructing such a compiler is to write a code as indicated by the flow diagram of Fig. 5-13. Then all the compiler need do is interpret the form in which a particular $f(x)$, x_0, and ϵ are introduced into the computer and put these values into the proper addresses of the code. Of course conventions must be made for the form of the particular $f(x)$, x_0, and ϵ, for the exit, and for the final location of the compiled code.

Simple extensions of such a compiler might enable it to compile loops within loops, where now some notation must be introduced to tell the compiler which loops are within which loops. We may make the compiler more versatile by enabling it to form partial products or partial

sums as well as successive iterations. Again perhaps more general functions than these can be introduced. The whole thing could be integrated and amalgamated with the algebraic compiler to act as one large compiler, and so forth.

5-9. Exercises

5-1. Write out precisely what an automonitor program would print if applied to the code of Table 3-1.

5-2. Suppose that a code contains a negative *jump* instruction. How can the automonitor program handle such an instruction? After the automonitor program executes the negative jump, how can it regain control?

5-3. Write a simple post-mortem debugging program as required for the third method of Sec. 5-2.

5-4. Write in symbolic code the series of instructions that lead to the setting up of the α address of the *transfer* instruction of subroutine Q in Fig. 5-3.

5-5. How would the evaluation of a function of two independent variables affect the discussion of Sec. 5-3?

With reference to the simulational interpretive routine discussed in Sec. 5-4:

5-6. Write out in detail the subroutine for multiplication, floating unnormalized.

5-7. Write out a subroutine for divide, floating unnormalized.

5-8. Write out the codes for boxes 3 to 6 of the flow chart of Fig. 5-4.

5-9. What other information must the format of the pseudoinstruction "Search" contain in addition to the "type" of the list being searched for? (HINT: How does the computer know where the first list is?)

5-10. Write an interpretive code for "Search," using the flow chart of Fig. 5-5 and a two-address relative indexed instruction format. (HINT: Use an *extract* instruction to isolate "type"; use a return jump to direct the interpreter routine exit to the next instruction of the main program.)

5-11. When the proper list has been found by the *search* pseudoinstruction, what record is made of its location and how can this record be used? (HINT: Remember the index register.)

5-12. What addresses must be specified by a pseudoinstruction that will compare two ordered lists and record the common numbers? (HINT: Three lists are involved: the two ordered lists and the final record.)

5-13. Draw a detailed flow chart of an interpretive routine that will interpret a *close up* pseudoinstruction.

5-14. Draw a detailed flow chart of an interpretive routine that will interpret a *make space* pseudoinstruction.

5-15. Deal a four-handed poker game. How would you record the results of the deal as a threaded list in a computer's memory?

5-16. Compare the contiguous-component and threaded-component list structures as far as using sublists is concerned.

5-17. Give an example of a subroutine that uses itself. How would a push-down list be utilized here?

5-18. Give an example of a subroutine that would use itself, and where this use would *again* entail its own use. How would a push-down list be used here? [HINT: In general a routine for a recursive (inductively defined) function uses itself as a subroutine.]

5-19. Write the specific code for the flow diagram of Fig. 5-11.

5-20. Write the code for box 2 of Fig. 5-10.

5-21. Draw a detailed flow diagram for the address-assigner compiler routine described in Sec. 5-7.

5-22. Using a one-address machine-language format, show by example that a right-to-left scan on an algebraic function results in a more efficient compiled code (with fewer instructions) than a left-to-right scan. [HINT: Use as an example the function $((A \cdot V)/(Y + Z))$.]

5-23. How would the flow chart of Fig. 5-12 be adjusted to account for a convention that enabled the omission of parentheses and utilized the rules that $\cdot$ and / supersede $+$ and $-$ [e.g., where $(a + (b \cdot c))$ is written as $(a + bc)$, and $(a - (b/c))$ is written as $(a - b/c)$]?

5-24. How would the flow chart of Fig. 5-12 be adjusted to account for numerical constants in the function?

5-25. Write a program for the simplified recursive compiler.

REFERENCES

Barton, R. S.: A New Approach to the Functional Design of a Digital Computer, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 393–396.

Bemer, R. W., H. J. Smith, and F. A. Williams: Design of an Improved Transmission Data Processing Code, *Communs. ACM*, vol. 4, no. 5, pp. 212–217, May, 1961.

Gelernter, H., J. R. Hansen, and C. L. Gerberich: A Fortran Compiled List-processing Language, *J. Assoc. Computing Mach.*, vol. 7, pp. 87–101, April, 1960.

Goodman, Richard: "Annual Review in Automatic Programming," Pergamon Press, New York, 1960.

Greenwald, I. D.: A Technique for Handling Macro Instructions, *Communs. ACM*, vol. 2, no. 11, pp. 21–22, November, 1959.

Hopper, G. M.: Automatic Programming; Present and Future Trends, *Proc. Symposium Mechanization of Thought Processes*, National Physics Laboratory, Teddington, England, 1959.

Kavanagh, T. F.: TABSOL—A Fundamental Concept for Systems-oriented Language, *Proc. Eastern Joint Computer Conf.*, December, 1960, New York, pp. 117–127.

McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computations by Machine, Part I, *Communs. ACM*, vol. 3, pp. 184–195, April, 1960.

McIlroy, M. D.: Macro Instruction Extensions of Compiler Languages, *Communs. ACM*, vol. 3, no. 4, pp. 214–220, April, 1960.

Miller, G. A.: Finite State Language, *Inform. & Control*, vol. 1, pp. 91–112, 1958.

Perlis, A. J., and C. Thornton: Symbol Manipulation by Threaded Lists, *Communs. ACM*, vol. 3, pp. 195–204, April, 1960.

Proceedings of the Fourth Annual Computer Applications Symposium, Armour Research Foundation, Oct. 24–25, 1957 (especially R. W. Bemer, The Status of Automatic Programming for Scientific Articles).

Wang, Hao: Toward Mechanical Mathematics, *IBM J. Research Develop.*, vol. 4, pp. 2–22, January, 1960.

CHAPTER 6

ALGEBRAICALLY ORIENTED LANGUAGES: ALGOL

6-1. Languages

Introduction. In this chapter we shall present an example of a complete, self-consistent automatic-programming language. In the previous chapter we have studied how the computer can be prepared or precoded with various automatic programs to aid the use of a subroutine, to translate symbolic codes, to transfer parts of a code, to handle lists automatically, to decode algebraic symbols, and to compile a loop code. There were essentially two parts to each of these discussions, the part that told *how to use* the automatic program in writing a code and the part that described how the automatic program itself was constructed. We have been building up to the idea of a general *automatic-programming language*, in which the procedures for *using* a combination of such automatic programs are unified and standardized, and for which the programs themselves are made compatible with each other.

Thus in this chapter we consider a new process, that of establishing the unified language itself. With the combinations of methods presented in the previous chapter, it is clear that a language can be devised that is independent of particular computer instructions. Hence a code can be written in terms of the *automatic language alone*, and specific computer instructions, the *machine language*, need never be mentioned. If automatic programs were written for different computers to interpret the same automatic language, then any code written in this language could be run equally well on any of the different computers.

As will be shown, establishing a complete and self-consistent automatic-programming language is far more complicated than just listing a set of pseudoinstructions. However, before we initiate our discussion several words of caution are in order. First, in this text our discussion will be of an informal nature, with such terms as "complete," "self-consistent," and even "language" not precisely defined. Although we shall continue to use terms that have particular meanings in the field of linguistics, we do not in any way intend to make, or even imply, a true language analysis. Rather we hope that the normal connotation of such words will assist the student in remembering their definitions as applied to

automatic-programming "languages." We shall occasionally use English as an analogy, but only as a pedagogic aid—no other connotation is meant in such instances. Even our discussion of automatic-programming languages will be from an elementary, introductory, descriptive point of view. We shall include only those concepts that are considered pedagogically necessary for the purposes at hand. Our discussions will be oriented directly toward the understanding of the methods involved in automatic programming, and no rigor will be intended. The astute reader, however, may at times sense the more basic ideas that are being touched upon. For a deeper insight into the basis of languages, of both the human and automatic-computer types, other, more detailed studies should be sought.†

With these qualifications in mind, it is appropriate for our purposes to consider *four ingredients* of a language,‡ namely, *characters*, *words*, *expressions*, and *statements*. These are really the hierarchical ingredients, because words are made from combinations of juxtaposed characters, expressions are composed of groups of words, and statements consist of specified combinations of expressions, together with certain words. The description of an automatic-programming language involves the specifications of these four ingredients: namely, (1) a display of the allowable characters, (2) rules about how to form words properly, (3) descriptions of the meaningful expressions of the language, and (4) discussions about the kinds of statements of which the language is made. The analogy of these structures to English is clear.

In an exposition of an automatic-programming language, the descriptions themselves can be classified into the following types: the *syntactical definitions*, the *semantic discussions*, and the *grammatical forms*. The syntactical definitions make explicit precisely what *structures* are to be meaningful in the language; these definitions are concerned with the *proper construction* of words, expressions, and statements. The semantic discussions are concerned with the *meanings* to be attributed to the various structures. The grammatical forms are concerned with the proper *relationships* among the different constructs, particularly the statements.

Characters, Words, Expressions, and Statements. The *characters* of a language are the basic, indivisible configurations in terms of which the language is written. For example, for English the character list includes the 26 upper-case (capital) letters, the 26 lower-case (small) letters, the 10 digits, the punctuation characters comma, semicolon, colon, period, space (i.e., blank), quotation marks, parentheses, and dash, and finally

† See, for example, N. Chomsky, "Syntactic Structures," Mouton & Co., The Hague, Netherlands, 1957.

‡ Henceforth we shall use "language" to mean "automatic-programming language" unless otherwise specifically stated.

the algebraic characters for the operations add, subtract, multiply, and divide and for the relations equals, greater than, and implies. (In actual practice this list of 77 characters used in English can be made much longer—see Exercise 6-1.) Similarly, for an automatic-programming language the characters that can be utilized must be listed and agreed upon—no other characters than those on the character list can be used or interpreted! The English analogy is typing: no other characters than those on the keyboard can be used when typing on a particular typewriter. Since codes must be typed or punched with particular *binary* patterns for input to the computer, the necessary limitation on the characters to be used in a language is easy to understand.

The choice of the *number* of different characters to be utilized in an automatic language is, of course, up to the individual who is constructing the language. With human written languages as an analogy, an important observation can be made concerning this choice. There are broadly three types of written languages: those in which each word is associated with a single character, called a pictograph, like (early) Chinese; those in which each character represents a syllable, and in which each word is made up of the characters representing the syllables, like Devanagari;† and those in which the characters form an alphabet and combinations of letters are used to make up the syllables and words. Over 100,000 characters are required for a pictographic-character language (see Fig. 6-1); several hundred are required for a syllabic-character language; relatively few are required for an alphabetic-character language (e.g., the basic 26 for English and 24 for Korean—see Fig. 6-1). The lesson to be learned is that the efficiency of an alphabetic-character representation goes up as the number of ideas to be expressed by the language goes up; similarly, it is easy to see that the efficiency of a pictographic language can be high when the number of ideas to be expressed is small. Of course, mixtures of alphabetic-character and word-character languages are ordinarily used. For example, the algebraic symbology used with English in mathematics texts is really a pictographic-character language, where for example x stands for the *unknown*, or 7 stands for *seven*. Even in mathematics, the difficulties of the word-character languages are sometimes felt, as we must frequently extend our list of symbols to include the Greek, German, Hebrew, etc., characters also. For automatic-programming languages, the list of available mathematical characters is usually limited to the letters, and hence it is common to use letter combinations for independent and dependent variables, such as **FUNCT** or **VAR**.

The *words* of a language are structures formed directly from the characters of the language and are usually the smallest units that have meaning by themselves. For example, in English a word (not necessarily an

† Devanagari is the basic written Indian language.

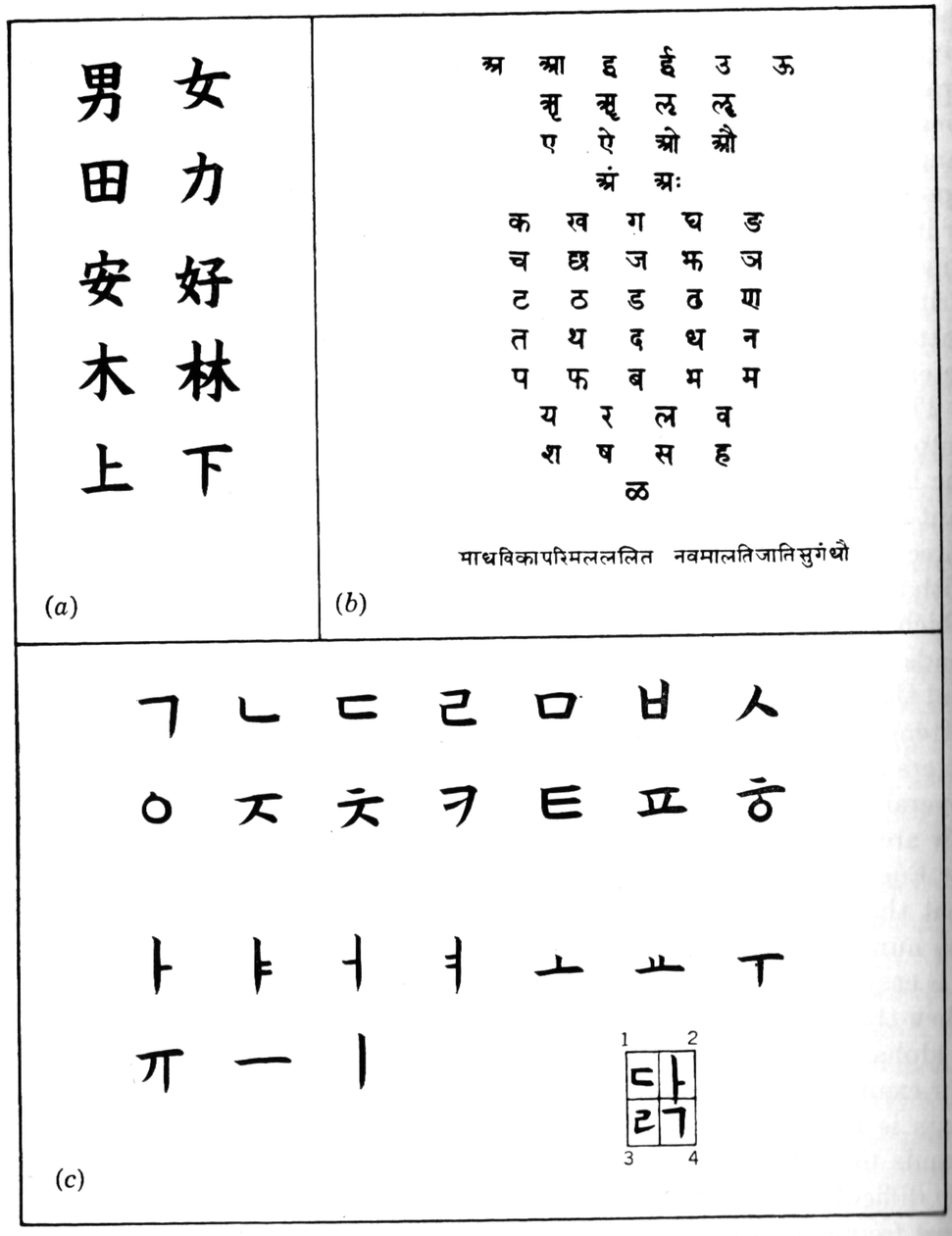


FIG. 6-1. Types of written languages. (*a*) Selected *pictographs* of the Chinese language. Top row: left, "man"; right, "woman." Second row: left, "rice field"; right, "strong." Note that "strong" with a rice field on top is a "man." Third row: left, "peace"; right, "good." Note that "peace" is a woman with a roof; similarly, "good" is a woman with a child. Fourth row: left, "tree"; right, "forest." Fifth row: left, "up"; right, "down." (*b*) Example of *syllabary* characters of Devanagari, the basic written language of many Indian people. The figure shows some of the characters above and some words on the line below. (*c*) The Korean *alphabet*. The first two rows represent the consonants; the next two rows represent the vowels. Words are formed by placing letters in a box. We have illustrated the four-letter word for "hen" at the lower right; the numbers indicate the order in which the letters are pronounced.

intelligible or "well-formed" word) is a combination of juxtaposed (side-by-side) letters followed by a space or punctuation character. In this same sense a number, which is a combination of juxtaposed digits, is also a "word." Although the idea of the juxtaposition of characters seems trivial to us, some languages, such as Korean, have their letters placed in arrays to form words (see Fig. 6-1). Of course in mathematics an array of numbers, to form, say, a matrix, is commonplace! However, because computer print-out mechanisms are usually designed to produce rows of characters, all automatic-programming "words" are composed of juxtaposed letters or digits. In such languages the "words" are frequently called *identifiers*, *variables*, *labels*, or *numbers*.

Expressions are the phrases or clauses of automatic-programming languages and are composed of words. In such languages, expressions are usually associated with some *value*. For example, "$a + bx$" would be an *arithmetic* expression, because if the values of a, b, and x were given, the value of $a + bx$ could be determined. Similarly "p or (q and not r)" would be a *Boolean* expression, because if the truth values of p, q, and r are given, the value of the expression could be calculated. Finally "if $a > b$ then EILEEN, otherwise ERIC" would be a *designational* expression, because it has the value EILEEN if $a > b$ and the value of ERIC if $a \leq b$. Thus expressions are usually identified as such because they consist of a group of words that have a central or unifying meaning, such as a "value." This is analogous to the identification of qualifying phrases or clauses in English.

A *statement* of an automatic-programming language is the smallest structure that represents a complete thought—or a complete description of a computation to be performed—and is directly analogous to a *sentence* of English or to an *instruction* of a machine language. Of course for an involved computation, many statements will be necessary, like a paragraph of English or a code of a machine language. However, just as there are a limited number of operations that instructions can perform for a particular computer, in an automatic-programming language there are a limited number of types of statements in the language.

In summary, then, our analogy with English has displayed a hierarchy of structures, in which the statements (sentences) are composed of expressions (phrases, clauses), the expressions are composed of identifiers, variables, labels, and numbers (words), and these in turn are composed of characters (letters, digits). To give precise definitions of meaningful or "well-formed" words, phrases, and sentences is difficult to do for English, or any human written language. This task for automatic-programming languages is, of course, much easier to do, and involves the concepts of syntax, semantics, and grammar that we shall now discuss.

The Syntactical Definitions. The *syntax* of a language consists of definitions of the structures to be formed from the characters of the language.

The concept of the *recursive definition* is frequently used in such syntactical descriptions of languages. Consider as an example a recursive definition of a number: The idea is to build up a number from the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, and from +, −, and the decimal point. We begin by defining

1. *An unsigned integer* as	(*a*) *a digit* or (*b*) *an unsigned integer followed by a digit*
2. *A decimal fraction* as	*a decimal point followed by an unsigned integer*
3. *A decimal number* as	(*a*) *an unsigned integer* or (*b*) *a decimal fraction* or (*c*) *an unsigned integer followed by a decimal fraction*

Finally we have the definition of

4. *A number* as	(*a*) *decimal number* or (*b*) + *decimal number* or (*c*) − *decimal number*

To see how this series of definitions works, let us build up the number −876.123 in this fashion. We would start with the

Unsigned integer 8	by applying definition 1*a*

From this we would successively form

Unsigned integer 87	by applying definition 1*b*
Unsigned integer 876	by applying definition 1*b*

Similarly we would form the unsigned integer 123 by successive applications of definition 1. Then we can form the

Decimal fraction .123	by applying definition 2

Next we form the

Decimal number 876.123	by applying definition 3*c*

Finally we can state that we have the

Number −876.123	by applying definition 4*c*

Of course, given a collection of digits, the definitions can be utilized in repeated application (in reverse) to determine what the collection is. For example, 256 is both a number (by definition 4*a*) and a decimal number (by definition 3*a*) and an unsigned integer (by definitions 1*a* and 1*b*).

The problem of describing a language involves a peculiarly unique difficulty. Distinction must be made during the discussion between the language being described and the language being used in making the description. If the language being described is called simply "the language," then the language in terms of which the description is being made is called "the metalanguage." Clearly the *metalanguage* and the *language* must be distinguishable from each other, or else much ambiguity and imprecision would result. In fact, many comedians utilize such ambiguity as the basis of their comic techniques: as in the telephone joke, where the operator requests the spelling of the name "Jones" and on receiving a reply starting "J like in Jack," then asks, "How do you spell Jack?"

Frequently a *meta*-metalanguage is required, for we consider the general names of the types of language structures to be in the metalanguage, and hence discussions about these names must be in the metametalanguage. We shall make the distinction between the language and metalanguage by utilizing *boldface* characters for the language. We shall not generally make any distinction between the metalanguage and the meta-metalanguage but will leave the proper meaning and interpretation to be obtained from the context.

Syntactical definitions can frequently be greatly shortened by utilizing a special meta-metalanguage symbolism. For example, let the symbol ::= represent "is defined as," the symbol | mean "or," and ⟨ ___ ⟩ indicate "the structure ___ of the language." Thus, for instance, definition 1 given above for an unsigned integer becomes

1. ⟨unsigned integer⟩ ::=	⟨digit⟩|⟨unsigned integer⟩⟨digit⟩

where juxtaposition of ⟨ · · ·⟩⟨ · · ·⟩ means "followed by." In this notation the rest of the definitions become

2. ⟨decimal fraction⟩ ::=	.⟨unsigned integer⟩
3. ⟨decimal number⟩ ::=	⟨unsigned integer⟩|⟨decimal fraction⟩ |⟨unsigned integer⟩⟨decimal fraction⟩
4. ⟨number⟩ ::=	⟨decimal number⟩ |**+**⟨decimal number⟩ |**−**⟨decimal number⟩

Note that if we use an actual symbol of the language itself, such as **.**, **+**, or **−**, then we do not need the brackets ⟨ ⟩, which are used only to denote generic names of the symbols of the language.

The Semantic Discussion. By *semantics* is meant the discussions of the *meaning of the structures that were syntactically defined.* Thus the semantic discussion associated with "number" would consider the cardinal and ordinal meanings of numbers and their uses. Four types of discussions are generally involved: discussions of the general role and purpose for the structure; of the kinds, or classification, of structures of that type; of the relation of the structure's function to its syntactical definitions;

and of the relation of the structure to its use in the higher grammatical forms in which it plays a part.

For example, if we were discussing the semantics of English words, we would first describe them as the indivisible basic parts of English that have content (i.e., can communicate ideas) in themselves. Thus a word is the smallest unit of English that has meaning when taken by itself. Second, we would describe the kinds of words, such as nouns, verbs, adjectives, adverbs, prepositions, etc. Third, we would discuss, say, the various endings that words can have from a syntactical point of view, with reference to the meaning of the words, such as that the addition of an *s* to a noun that denotes an object will make the noun plural. Finally, we would discuss, say, the fact that nouns have particular places in sentences, as subjects, objects, etc. Of course our English illustration is manifestly naïve if it is taken to apply literally to the English language, but we do feel that we have illustrated, if only in a superficial way, the role the *semantic discussion* plays in the total formulation and description of an automatic-programming language.

The Grammatical Forms. Consideration of grammatical forms applies mainly to the statements of the automatic-programming language (or the sentences of English) and includes discussions of both the general kinds of statement forms and the relationships of statements, one with another. Specific discussions of these must be relegated to the sections on the specific automatic-programming languages themselves; in this section we shall discuss only the general considerations of grammatical forms. Note, however, that by "grammatical forms" we really mean the syntax and semantics of combinations of statements in relation to each other. The automatic-programming statement, like the English sentence, is the smallest structure by which a complete thought can be expressed; hence consideration of the types of meanings, and use of such statements in relation to other statements, is of foremost importance.

Automatic-programming languages are of course all designed for a specific purpose, to enable the programmer more easily to tell the computer *what to do.* Hence analogy to all of the English language is not really valid; rather the analogy should be made to that part or type of English utilized in cookbooks, or other "what to do" (or, if you will, "how to do it") manuals. Hence in our explanatory analogy of grammatical forms, we shall limit ourselves to forms found in such manuals. In fact, to describe the kinds of grammatical forms (and also to add a little flavor to the discussion) we shall utilize the analogy of an actual cookbook recipe for making chocolate fudge, which follows:†

† Adapted from "The Good Housekeeping Cook Book," p. 752, Rinehart & Co., New York, 1949.

2 cups granulated sugar
1 cup bottled milk
½ teaspoon salt
2 oz unsweetened chocolate
2 tablespoons white corn syrup
2 tablespoons butter
½ teaspoon vanilla flavoring

(NOTE: *Cool* means to let stand at room temperature until no longer warm to the touch. *Stir* means to blend ingredients with a circular motion of the spoon, widening the circles until all ingredients are well blended.)

Combine first 5 ingredients in saucepan over low heat. Stir constantly until sugar dissolves. Then, stirring occasionally, cook gently until a little mixture, dropped in cold water, forms a soft ball. Remove from heat; drop in butter. Let stand without stirring until you can just rest hand on bottom of pan. Add vanilla; beat with spoon until mixture loses gloss and small amount, dropped from spoon, holds its shape. Turn into a greased 9- by 5- by 3-in. pan. Cool; cut in squares.

Observe that the recipe is naturally divided into two parts: the first, a *declaration* of what will be needed in the action; and the second, the list of *procedures* to be executed, in order, in performing the task. Analogously, automatic-programming languages contain two types of grammatical forms, called *declarations* and *procedures.* The declarations are of two types: the *data declarations,* or records of the items to be utilized during the execution of the procedures (e.g., the list of ingredients), and the *procedure declarations,* or subroutines (e.g., the general processes of cooling and stirring, as described in the recipe) that are referred to in the actual *procedure statements* of the procedures. The procedures to be directly executed during the execution of the recipe are analogous to the instructions of an ordinary machine-language code, considered in Part 1 of this text. The *procedure statements* describe the operations to be performed on the ingredients, or data, referring to the subroutines of the declaration part of the recipe.

There are many types of procedure statements, some of which may involve decisions—such as "Stir constantly until" sugar dissolves. The main field for which the language was developed, or its orientations, becomes apparent on examining the actions to be performed by the collection of procedure statements that are used in the language. Thus "combine," "stir," "cool," and "beat" are characteristic of cooking, while "assign the value to," "take the square root of," and "repeat the following computations until" are characteristic of algebraic computations.

In Fig. 6-2 we have summarized some of the above remarks. Note that the procedure declarations are composed of statements. Note also that we can quite naturally extend the meaning of "expressions" to include "phrases and clauses" as well as words.

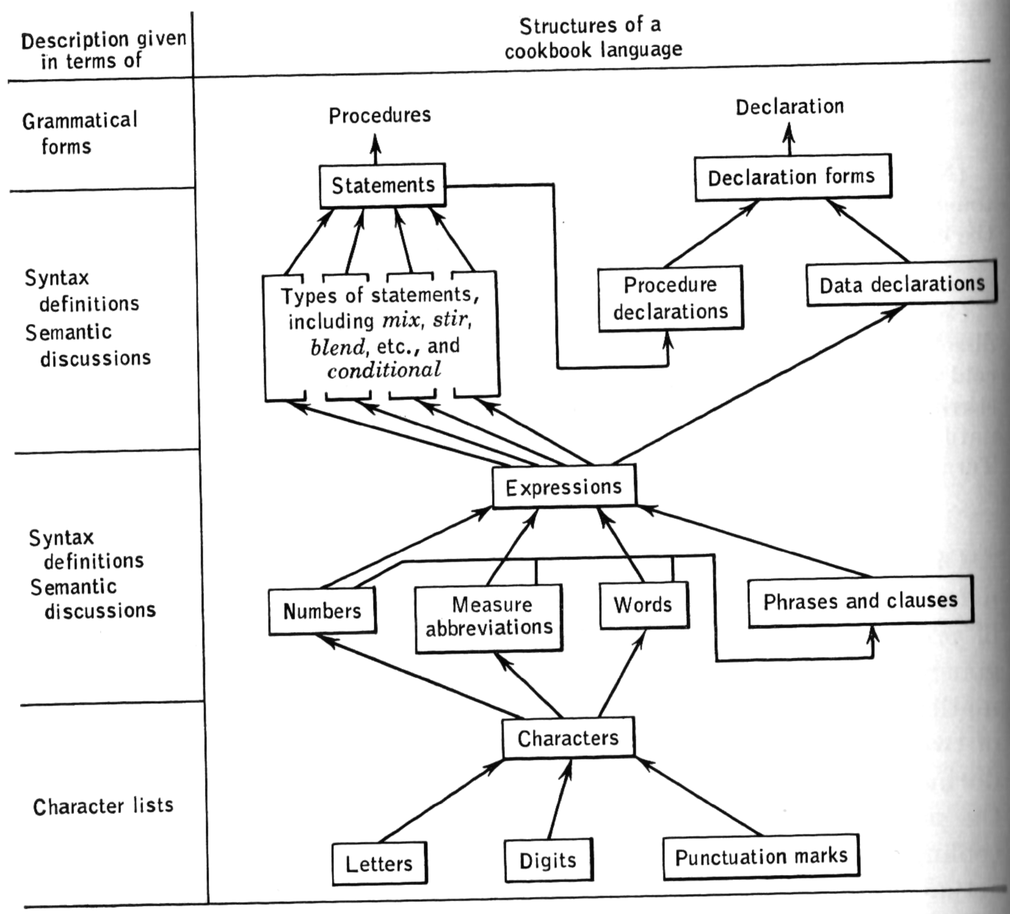


FIG. 6-2. Summary of structures of a "what to do" or "cookbook" language. Note that the declaration forms are of two types: (1) a list of ingredients, or data, and (2) descriptions or subroutines of processes, which latter are given in terms of statements. The statements and the data declarations are composed of expressions, which can be formed directly from numbers, words, and measure abbreviations or indirectly through phrases and clauses. Finally the expressions are composed of combinations of characters.

6-2. Example of an Algebraically Oriented Language: ALGOL

Purpose. As we have seen in the previous section, all automatic-programming languages are composed of *expressions* and *statements*, which are analogous, respectively, to the *words* and *sentences* of English, or to generalized *addresses* and *instructions* of direct machine-instruction systems (i.e., machine languages). The form and purpose, respectively, of the allowable expressions and statements are described in terms of *syntax* and *semantics*, and the rules governing the use of statements to form codes or programs are discussed under the *grammatical forms*. All languages can be interpreted in terms of such an organization; the differences between languages are found in the precise syntactical, semantic, and grammatical entities defined for the different languages, and these definitions in turn depend on the purposes for which the different languages were constructed.

The purpose of the particular language now to be considered is to assist in the programming of primarily algebraic numerical computations. This language is called ALGOL, which stands for *algo*rithmic *l*anguage, and has been generally accepted internationally as a common algebraically oriented language. As noted above, a program written in terms of a particular language can be run on any computer for which there is an automatic program that can compile or interpret the language. This fact indicates the feasibility of the concept of establishing a universally acceptable international automatic language, analogous to the universal written language of music.

Toward this goal, a series of conferences and meetings were held during 1958 and 1959, culminating in a meeting in Paris on Jan. 11–16, 1960, among representatives from the United States, Denmark, England, France, Germany, Holland, and Switzerland. As a result of the 1960 meeting, the "Report on the Algorithmic Language ALGOL" was published by the Association for Computing Machinery.† In this and the following sections we shall present the basic concepts included in ALGOL. Wherever pedagogical considerations warrant, we shall utilize simplified versions of particular structures of ALGOL. For the complete language, the reader is directed to the reference given in the footnote. ALGOL was chosen as an example of an algebraically oriented language because it includes almost all the basic concepts of other algebraic languages and its use in application as well as in publications is rapidly becoming widespread.

Example of the Language. Since the purpose of ALGOL is to describe computational processes, the *arithmetic expression*, together with the so-called *assignment statement*, is of central importance. The arithmetic expression is composed of combinations of the arithmetic operations with numbers, variables, and functions; the assignment statement, containing within it an arithmetic expression, tells the computer to proceed with the performance of the computations indicated by that arithmetic expression. As an illustration of how the language works, consider the following simple example of a code to compute a root of $Ax^2 + Bx + C = 0$ for $A = +3$, $B = +1.7$, and $C = -0.31$. One of the two roots is given by the well-known formula

$$x = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

† See *Communs. ACM*, vol. 3, no. 5, pp. 299–314, May, 1960.

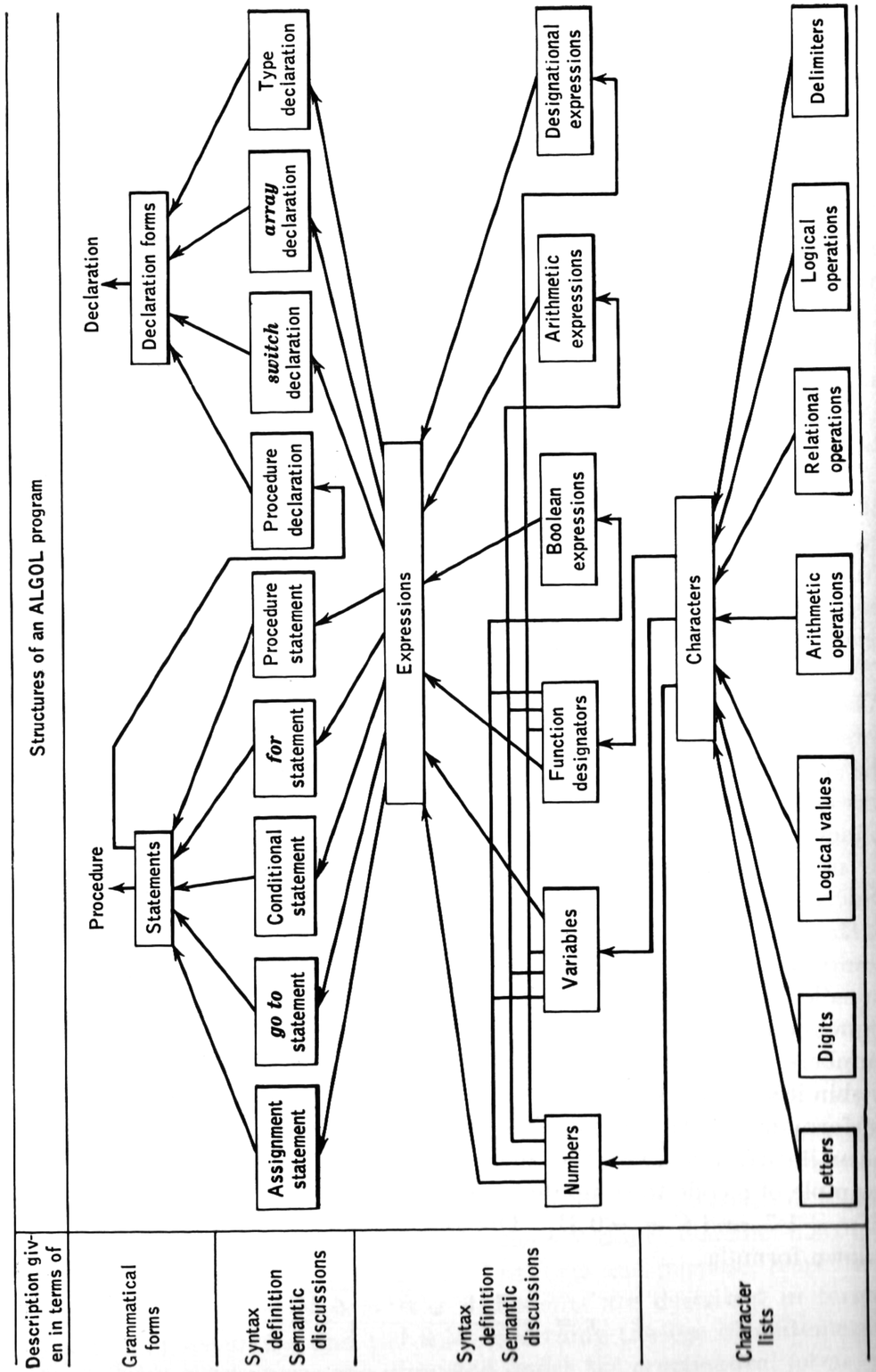


Fig. 6-3. "Anatomy" of a language, showing the hierarchy of the structure of ALGOL.

The code to evaluate the right-hand side of this equation becomes†

; A :=3; B :=1.7; C :=−0.31;
***root*:=(−B+*sqrt*(B↑2−4×A×C))/(2×A); *stop*;**

The meaning of this jumble of symbols becomes more or less obvious with the following explanation: First the symbols between semicolons (; . . . ;) are *statements*. This is analogous to putting instructions, constants, etc., on the separate lines of a coding sheet, or to placing English sentences between periods. Statements are normally executed in sequence. The first three statements are assignment statements and record the desired values of A, B, and C; the fourth statement is also an assignment statement and computes the root; the last statement stops the routine. In this example, the *expressions* are **A**, **3**, **B**, **1.7**, **C**, **−0.31**, ***root***, ***sqrt*(B↑2−4×A×C)**, and ***stop***; here **B↑2−4×A×C** is an arithmetic expression, or the whole collection

(−B+*sqrt*(B↑2−4×A×C))/(2×A)

can be considered an arithmetic expression. The numbers involved are **3**, **1.7**, **−0.31**, **2**, and **4**; the variables are **A**, **B**, ***root***, and **C**; the function involved is ***sqrt***; and **−**, **↑**, **/**, and **×** are the arithmetic operations. The automatic program that is to interpret this language compiles the appropriate instructions and, for the square root (***sqrt***), refers to the appropriate subroutine.

In addition to the arithmetic expression and the assignment statement, many kinds of nonarithmetic expressions and other statement types must be defined, to describe *alternatives* or *iterative repetitions* of assignment or computing statements. Also, special declarations must be defined (just as part of the fudge declaration consisted of special forms of "sentences" listing the ingredients). Altogether, a reasonably complicated structure is developed. Figure 6-3 summarizes the main types and the hierarchical structure of the statements, declarations, and expressions, all of which will be considered in this and the next sections.

Characters. There are seven types of characters included in the ALGOL language: namely, (1) letters, (2) digits, (3) logical values, (4) arithmetic operation symbols, (5) relational operation symbols, (6) logical operation symbols, and (7) delimiters. To be concrete, we can list the different types of characters, utilizing our metalanguage notation given above:‡

† Although ALGOL does not have a ***stop*** statement, we shall use it from time to time for pedagogical reasons.

‡ In this chapter we shall also use some Greek letters occasionally, for our convenience. Further, we shall use italic letters for standard ALGOL forms.

⟨letter⟩ : : = **a**|**b**|**c**|**d**|**e**|**f**|**g**|**h**|**i**|**j**|**k**|**l**|**m**|**n**|**o**|**p**|**q**|**r**|**s**|**t**|**u**|**v**|**w**|**x**|**y**|**z**
|**A**|**B**|**C**|**D**|**E**|**F**|**G**|**H**|**I**|**J**|**K**|**L**|**M**|**N**|**O**|**P**|**Q**|**R**|**S**|**T**|**U**|**V**|**W**|**X**|**Y**|**Z**
⟨digit⟩ : : = **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**
⟨logical value⟩ : : = **T**|**F**
⟨arithmetic operator⟩ : : = +|−|×|/|↑
⟨relational operator⟩ : : = <|≦|=|≧|>|≠
⟨logical operator⟩ : : = ≡|⊃|∨|∧|¬
⟨delimiter⟩ : : = (|)|[|]| :|.|;|,|:=

First, let us make an observation about our notation: we did not need to use the brackets ⟨ ⟩ on the right-hand sides because the boldface characters are themselves in the ALGOL language. The "logical values," **T** for true and **F** for false, must be distinguished from the *letters* **T** and **F** by the context. The arithmetic operator ↑ means "raise to the power"; thus A^2 would be written **A↑2**. The logical operators differ from the symbols used in Chap. 10 of this text; the correspondence is as in Table 6-1.

TABLE 6-1. LOGICAL-OPERATOR SYMBOL NOTATION

Meaning	ALGOL	Chapter 10
Truth-value equivalence	≡	=
If, then (implies, is included in)	⊃	→
Or	∨	+
And	∧	·
Not A	¬**A**	$\bar{A}$

Expressions. We shall distinguish six types of expressions, namely, (1) numbers, (2) variables, (3) function designators, (4) Boolean expressions, (5) arithmetic expressions, and (6) designational expressions. We shall be concerned with the first three in the remainder of this section and with the last three in Sec. 6-3.

If **a** represents a digit, then a *number* can be an integer **aa · · · a**, a decimal fraction **.aa · · · a**, or an exponent part $_{10}\pm$**aa · · · a**, or any combination of these, signed or unsigned. For example, **6**, **−4711**, **+137.06**, **2.999**$_{10}$**12** ($= 2.999 \times 10^{12}$), $_{10}$**−12** ($= 10^{-12}$), and **3**$_{10}$**−12** ($= 3 \times 10^{-12}$) are all numbers. In ALGOL, a number can only act as the contents of an address, i.e., a value, and not the address itself, i.e., *not* a name or identifier. The syntactical definition of a number was given above as an illustration of the use of the metalanguage.

Variables are essentially the basic words of ALGOL and are the *names* (analogous to *addresses*) for numbers, scalar quantities, or logical values, as well as for "locations" of statements that can (eventually) be evaluated to result in a number or logical value, as used in ordinary or Boolean algebra. Two kinds of variables are distinguished, the *simple variable*

and the *subscripted variable.* A simple variable can be a single letter, or it can be a combination of several adjacent letters and numbers, but the first (leftmost) character must be a letter in order that simple variables and numbers can be distinguished. Thus, for example, **b**, **P63**, **Q382T**, and **ALPHA** are all simple variables. In our metalinguistic notation we can write

⟨simple variable⟩ : : = ⟨letter⟩|⟨simple variable⟩⟨letter⟩
|⟨simple variable⟩⟨digit⟩

Note that, by recursive reasoning, according to this syntactical definition the simple variable must have a letter as its leftmost character.†

Subscripted variables designate values of quantities that are components of multidimensional arrays, such as the components of a list, a vector, or a matrix, etc. What would normally be the subscripts are placed in brackets. Thus a vector V_i would be written as **VECTOR[i]**, a matrix M_{ij} as **MATRIX[i,j]**, and so forth. The "subscripts" themselves may appear as integral numbers, simple variables, subscripted variables, or arithmetic expressions (see below). Note that the subscripts are intrinsically integer-valued, no matter in what form they appear. The comma distinguishes the different subscripts: for example, **VAR[6,ALPHA,3×i+k]** is a three-subscripted variable (see Exercise 6-11). Finally we have

⟨variable⟩ : : = ⟨simple variable⟩|⟨subscripted variable⟩

To identify *single-valued* (logical as well as numerical) functions by the usual algebraic functional notation, *function designators* are used. The independent variables are enclosed in parentheses and separated by commas, for example, **FUNCT(x,y,z)**. Here **FUNCT** (that is, the function name) is the same syntactical form as a simple variable and can be thought of as essentially the first address of a subroutine (i.e., *procedure declaration*) that can evaluate the function; for this reason **FUNCT** is called a *procedure identifier.* The **(x,y,z)** is the ordered set of independent variables, the numerical (or logical) values of which must be ascertained before the function is evaluated. The independent variables themselves

† For pedogogical reasons we have not utilized the *identifier* expression defined for ALGOL. The syntactical definition is

⟨identifier⟩ : : = ⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩⟨digit⟩

the same as that for a simple variable. In addition, we differ from ALGOL by choosing to classify *numbers* under expressions, omitting *strings*, and neglecting the syntactical definitions of *sequential operators, declarators,* and *specificators.* Once a grasp of ALGOL is gained from the present treatment, these slightly complicating features can easily be reviewed by the reader, as no new concepts are involved. See the reference of the footnote on p. 203 for the full ALGOL 60.

can be simple variables, subscripted variables, or arithmetic, Boolean, or designational expressions (see below). A syntactical definition of function designators would be†

⟨procedure identifier⟩ : : = ⟨letter⟩|⟨procedure identifier⟩⟨letter⟩
|⟨procedure identifier⟩⟨digit⟩

⟨independent variable⟩ : : = ⟨simple variable⟩|⟨subscripted variable⟩
|⟨arithmetic expression⟩|⟨Boolean expression⟩
|⟨designational expression⟩

⟨independent-variable list⟩ : : = ⟨independent variable⟩
|⟨independent-variable list⟩, ⟨independent variable⟩

⟨function designator⟩ : : = ⟨procedure identifier⟩ (⟨independent-variable list⟩)
|⟨procedure identifier⟩

Note that a function designator does not necessarily have an associated list of independent variables. An example might be **R**, the procedure identifier for choosing a random number. Certain function names should be reserved for the standard functions, such as ***abs*(N)** for the absolute value of a variable N, ***sign*(N)** for the sign (that is, + or −) of the value N, ***sqrt*(N)** for the square root of N, ***sin*(θ)** for the sine of θ, and so forth.

6-3. Boolean, Arithmetic, and Designational Expressions

Boolean Expressions. A *Boolean expression* is a rule for computing a *logical value*, that is, **T** or **F** (see Sec. 10-2 for a discussion of definitions of the Boolean operations; Table 6-1 gives their correspondence with ALGOL logical operation symbols). Table 6-2 summarizes the truth-value definitions of the operations.

TABLE 6-2. TRUTH-VALUE DEFINITIONS OF BOOLEAN OPERATIONS

a	F	T	F	T
b	F	F	T	T
¬a	T	F	T	F
a ∧ b	F	F	F	T
a ∨ b	F	T	T	T
a ⊃ b	T	F	T	T
a ≡ b	T	F	F	T

For example, if **a** is **T**, **b** is **F**, **c** is **T**, and **d** is **F**, then

$$((a \vee b) \vee (b \wedge c)) \equiv ((b \vee d) \supset (a \wedge c))$$

† Simplified from ALGOL 60.

on successive evaluation becomes

$$(T \vee F) \equiv (F \supset T)$$
$$T \equiv T$$
$$T$$

The logical variables, in our example **a**, **b**, **c**, and **d**, could of course represent *relations* whose values are true or false. For instance, **a** could represent $(x \geqq y)$, which if **x** is **3** and **y** is **2** would be true, i.e., would have the value **T**. In fact, in this relation **x** could itself be an arithmetic expression (see below), such as $u \times w + v$, as could **y** (see Exercises 6-12 through 6-15). On the other hand, the logical variables could themselves represent other logical expressions. Such logical expressions, which can include relations between arithmetic expressions and also other logical expressions, are called *simple Boolean expressions.* For example,

$$(((u \times w + v) \geqq y) \wedge ((u \times v) < z)) \vee ((a \vee b) \equiv (\neg c \supset d))$$

is a simple Boolean expression. If a convention is made on the rules of precedence for the operations, then many of the parentheses can be omitted. The precedence rules of ALGOL are as follows:

First.	Arithmetic expressions (according to rules to be given below)
Second.	$< \leqq = \geqq > \neq$
Third.	$\neg$
Fourth.	$\wedge$
Fifth.	$\vee$
Sixth.	$\supset$
Seventh.	$\equiv$

With these rules, the above simple Boolean expression can be written as

$$u \times w + v \geqq y \wedge u \times v < z \vee (a \vee b \equiv \neg c \supset d)$$

A metalinguistic syntactical definition of a simple Boolean expression that automatically includes such rules of precedence can be given as follows:

⟨relation⟩ : : = ⟨arithmetic expression⟩ ⟨relational operator⟩ ⟨arithmetic expression⟩

⟨Boolean term⟩ : : = ⟨logical value⟩ | ⟨variable⟩|⟨relation⟩
| ⟨function designator⟩ | (⟨simple Boolean expression⟩)

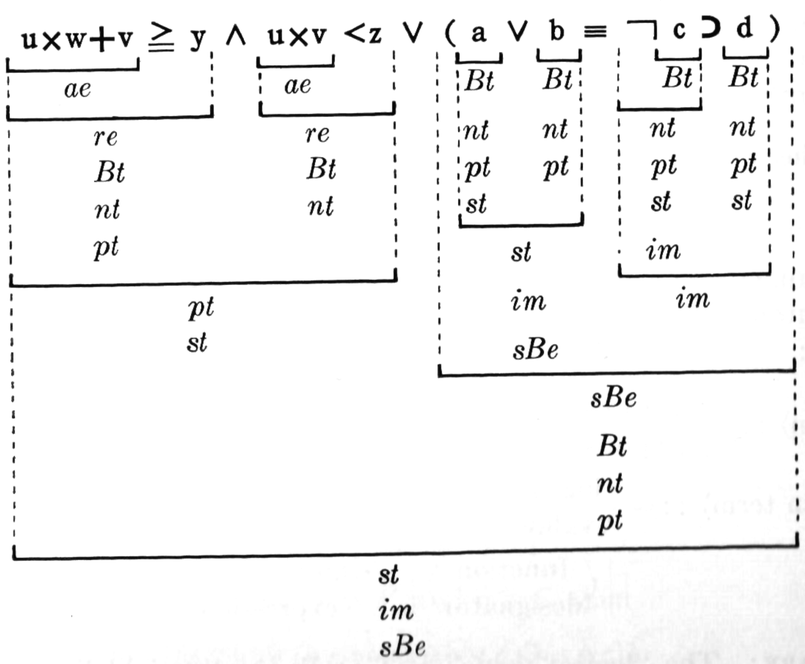
REMARK: The "logical value" is **T** or **F**; the "variable" must have a logical value, i.e., must be a logical variable; the function denoted by

the "function designator" must have a logical value, i.e., must be a logical function.

⟨neg-term⟩ : : = ⟨Boolean term⟩|¬⟨Boolean term⟩
⟨product-term⟩ : : = ⟨neg-term⟩|⟨product-term⟩∧⟨neg-term⟩
⟨sum-term⟩ : : = ⟨product-term⟩|⟨sum-term⟩∨⟨product-term⟩
⟨implication⟩ : : = ⟨sum-term⟩|⟨implication⟩⊃⟨sum-term⟩
⟨simple Boolean expression⟩ : : = ⟨implication⟩ | ⟨simple Boolean expression⟩≡⟨implication⟩

An explanation is in order! Note that each line from the "neg-term" on introduces another logical operator, in the order of precedence given above. Thus, at each level, the operations previously introduced must already have been introduced into the expression, i.e., must already have been evaluated. But the right-hand side of each line has two parts, the first of which repeats the previously defined level; this is to take account of simple Boolean expressions that do not involve that logical operation. Finally, the basic "Boolean term" includes as a possibility a "simple Boolean expression" in parentheses; this is to account for terms which the parentheses indicate are to be computed out of the precedence order.

As an example, consider the following syntactical analysis, where the abbreviations are *ae* for arithmetic expression, *Bt* for Boolean term, *re* for relation, *nt* for neg-term, *pt* for product-term, *st* for sum-term, *im* for implication, and *sBe* for simple Boolean expression:



Note that each symbol or combination of symbols must be developed

up to the point where it can be combined with other symbols. Also, we first had to show that **a∨b≡¬c⊃d** was a simple Boolean expression before it could be incorporated into the entire analysis as a Boolean term (see Exercise 6-17).

One further concept associated with a Boolean expression still remains to be introduced. This concept is concerned with the Boolean expression whose value itself depends on certain logical conditions and involves the so-called *if clause* of the form

⟨if clause⟩ : : = ***if*** ⟨Boolean expression⟩ ***then***

The "if clause" is used in connection with a whole expression of the form

if ⟨Boolean expression⟩ ***then*** ⟨simple Boolean expression⟩
else ⟨Boolean expression⟩

Such an expression is evaluated as follows: If the first Boolean expression is true, then the value of the whole expression is the same as that of the simple Boolean expression; otherwise the value of the whole expression is the same as the value of the last Boolean expression. That is, the value of the whole expression is the value of the simple Boolean expression, *or else* it is the value of the last Boolean expression, depending on the truth of the "if clause." With this in mind, we can make this definition:

⟨Boolean expression⟩ : : = ⟨simple Boolean expression⟩
|⟨if clause⟩ ***else*** ⟨Boolean expression⟩

For example, consider the Boolean expression

if **k<1** ***then*** **S>W** ***else*** **h≦c**

To evaluate this, we find the value of **k**, say **k = −3**, for which **k<1** is true; then we evaluate the relation **S>W** to find the *value of the Boolean expression.* Suppose, however, that **k = +3**: then **k<1** is false, and we evaluate the relation **h≦c** to find the value of the Boolean expression.

ONE FINAL REMARK: In the definition of the "Boolean term" given above, we should really have had "Boolean expression" in the parentheses, rather than the more restrictive "simple Boolean expression."

Arithmetic Expressions. An *arithmetic expression* is a rule for computing a *numerical value.* We have been building up to this important concept, whose full definition involves the "if clause" and hence the evaluation of a Boolean expression. First, however, we shall consider the part of the arithmetic-expression definition that does not involve ***if***, the *simple arithmetic expression.* This type of expression consists of combinations of simple and subscripted variables, functions, and num-

bers, with the arithmetic operations of **+**, **−**, **×**, and **/** representing addition, subtraction, multiplication, and division, respectively. An upward arrow **↑** represents exponentiation, i.e., raising to a power. Thus, for example,

2↑3 means 2^3
and **2↑3↑4** means $(2^3)^4$
while **2↑(3↑4)** means $2^{(3^4)}$

For **p** an integer **a↑p** is interpreted as the **p**th power of **a**; for **p** noninteger but **a** positive, we quite naturally have

$$\mathbf{a\uparrow p} = e^{(\mathbf{p}\times\ln(\mathbf{a}))}$$

Certain precedences hold that can permit the omission of parentheses, as follows:

First. **↑**
Second. **×**, **/** (called *mult operators*)
Third. **+**, **−** (called *adding operators*)

Unless otherwise indicated by parentheses, the evaluation of a simple arithmetic expression is made from left to right according to the precedences. Thus

+a−b/c×6/(p−q)↑2×u/v

means

$$(\mathbf{a} - ((((\mathbf{b} \times (\mathbf{c}^{-1})) \times 6) \times ((\mathbf{p} - \mathbf{q})^{-2})) \times \mathbf{u}) \times (\mathbf{v}^{-1}))$$

As in the case of the Boolean expression, we can produce a set of syntactical definitions that take account of the precedence order for arithmetic operations:

⟨term⟩ : : = ⟨unsigned number⟩|⟨variable⟩|⟨function designator⟩|(⟨simple arithmetic expression⟩)

⟨factor⟩ : : = ⟨term⟩|⟨factor⟩↑⟨term⟩

⟨mult-factor⟩ : : = ⟨factor⟩|⟨mult-factor⟩⟨mult operator⟩⟨factor⟩

⟨simple arithmetic expression⟩ : : = ⟨mult-factor⟩|⟨adding operator⟩⟨mult-factor⟩

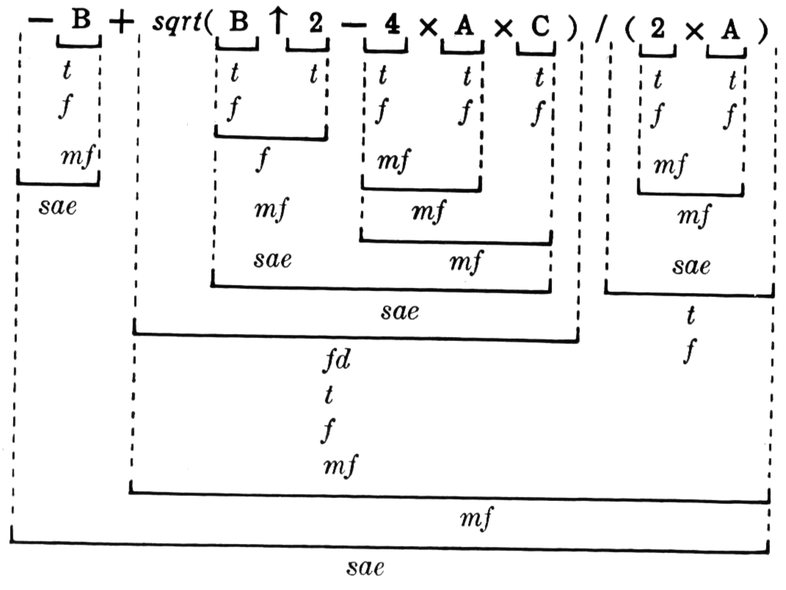
|⟨simple arithmetic expression⟩⟨adding operator⟩⟨mult-factor⟩

Notice that, in the definition of "term," an "unsigned number" was used. This is done to avoid the occurrence of, for example, **a+−b** for **a+(−b)**. Then in the definition of "simple arithmetic expression" one of the alternatives is

⟨adding operator⟩⟨mult-factor⟩

which puts a sign in front of the number. In this way only **(−b)** can become a "term," and not **−b** alone.

As an example, let us consider the following syntactical analysis, where *t* stands for term, *f* for factor, *mf* for mult-factor, *fd* for function designator, and *sae* for simple arithmetic expression:



Note that we first had to show that **B↑2−4×A×C** was a simple arithmetic expression in order to find that ***sqrt*(B↑2−4×A×C)** was a function designator. Then the function designator had to be redefined as a term, a factor, and a mult-factor before it could be used to combine with the factor **(2×A)**. Similarly **2×A** had to be worked up to a simple arithmetic expression before it could in turn take on the parentheses to become the term **(2×A)** (see also Exercise 6-29).

The variables and function designators that participate in the arithmetic computations naturally must take on numerical (as opposed to logical) values. It should be observed that in the evaluation of a simple arithmetic expression the actual numerical value of a term is computed as follows: For the unsigned number, it is the value of the number; for the variable, it is the current value most recently assigned in this phase of the computation; and for function designators, it is the value generated by the subroutine for the function (i.e., from its declaration—see below) when applied to the current value of the independent variables involved.

We come finally to the "*arithmetic*" *expression* itself, which includes the simple arithmetic expression as well as the "if clause" alternatives:

⟨arithmetic expression⟩ : : = ⟨simple arithmetic expression⟩
|⟨if clause⟩⟨simple arithmetic expression⟩
else ⟨arithmetic expression⟩

As an example of the evaluation of such an arithmetic expression, consider

if **Q>0** ***then*** **S+3×Q** ***else*** **3×S+Q**

If **S = 4** and **Q = +1**, that is, if the relation is true, then the value of the arithmetic expression is

$$\mathbf{S+3\times Q = 4+3 = 7}$$

On the other hand, if **Q = −1**, that is, if the relation is false, then the value of the arithmetic expression is

$$\mathbf{3\times S+Q = 12-1 = 11}$$

Since the last arithmetic expression in the definition can also involve an if clause, and so on, we could have the following as an arithmetic expression:

if **Q>0** ***then*** **S+3×Q** ***else if*** **S>5** ***then*** **3×5+Q**
else if **Q+S<0** ***then*** **S** ***else*** **Q**

From our definition it follows that, to evaluate such an expression, the "if clauses" are evaluated one by one in sequence from left to right, until one having the value T is found: the value of the arithmetic expression is then the value of the first arithmetic expression following this "if clause."

ONE FINAL REMARK: In the definition of "term" given above, we really should have had "arithmetic expression" in the parentheses, rather than the more restrictive "simple arithmetic expression."

Designational Expressions. Just as instructions in machine language are denoted by their *addresses*, so in an automatic-programming language statements are denoted (or named) by means of *labels*. We shall delve further into this in the next section when we discuss statements. However, the concept of a label is important to us here because a *designational expression* is a rule for obtaining a label from (usually) several possible alternative labels. Before we describe designational expressions in detail, it will be helpful if we first mention briefly how they are used in the ***go to*** *statement*. The ***go to*** *statement* is directly analogous to a *jump instruction* and appears as

go to ⟨designational expression⟩

whence the designational expression selects the label of the statement to which the jump is to be made. Designational expressions can also be used in conjunction with the ***switch*** declaration, but we shall defer consideration of this use until we discuss declarations.

The syntactical definition of a designational expression is

⟨label⟩ : : = ⟨variable⟩|⟨unsigned integer⟩

$$\left\langle \begin{matrix}\text{simple}\\ \text{designational}\\ \text{expression}\end{matrix} \right\rangle ::= \langle\text{label}\rangle \mid (\left\langle \begin{matrix}\text{designational}\\ \text{expression}\end{matrix} \right\rangle)$$

$$\left\langle \begin{matrix}\text{designational}\\ \text{expression}\end{matrix} \right\rangle ::= \left\langle \begin{matrix}\text{simple}\\ \text{designational}\\ \text{expression}\end{matrix} \right\rangle \mid \langle\text{if clause}\rangle \left\langle \begin{matrix}\text{simple}\\ \text{designational}\\ \text{expression}\end{matrix} \right\rangle \textit{else} \left\langle \begin{matrix}\text{designational}\\ \text{expression}\end{matrix} \right\rangle$$

For example, we might have

if **a>b** ***then*** **PDQ** ***else if*** **c>d** ***then*** **QRS** ***else*** **136**

Here if **a>b** is true, the label is **PDQ**; otherwise **c>d** must be evaluated. Then if **c>d** is true, the label is **QRS**; otherwise the label is **136**. As another example, consider

if **a>b** ***then*** (***if*** **c>d** ***then*** **QRS** ***else*** **136**) ***else*** **XYZ**

Here if **a>b** is true, then we must evaluate **c>d**; if **c>d** is true, the label is QRS, otherwise **136**. If **a>b** is false, then the label is **XYZ**.

6-4. Statements

Compound Statements and Blocks. Up to now we have been considering the kinds of structures of which statements are composed. Now we shall consider the statements themselves (cf. Fig. 6-3), which are the operational or functional structures of the language. Statements in an automatic-programming language are analogous to the instructions of machine language. The performance or execution of each statement is the action that carries out the desired computations; a sequence of statements is generally executed consecutively unless otherwise specifically indicated. Five types of statements will be considered: (1) *assignment statements*, which perform calculations as indicated by arithmetic or Boolean expressions; (2) ***go to*** *statements*, which can break the normal statement-execution sequence according to alternatives set forth by designational expressions and hence afford a means for jumping to other parts of a program; (3) *conditional statements*, which can place conditions on the omissions or inclusions of other statements in the statement-execution sequence; (4) ***for*** *statements*, which automatically carry out loop or recursive processes; and (5) *procedure statements*, which serve to invoke (call for) the execution of a procedure declaration, or subroutine. The first four of these will be considered in this section; discussion of the

fifth will be deferred until the declarations of the next section have been described.

Before initiating the detailed discussion of these statement types, however, we must first consider the ideas of *labels*, *compound statements*, and *blocks*. Let us recall from our initial example that statements are set apart by semicolons. Thus the statement **; S;** means "Execute the computations called for by **S**." It is often desirable to give a statement a label, or name, **L**, analogous to the address of an instruction. From the syntactical definition in the previous section, we see that the label **L** may be a simple or subscripted variable or an unsigned integer. If **L** is to be the name of statement **S**, we write **; L :S;**. Thus **; L :S;** tells the computer that the label **L** has been assigned to the statement **S** (and also that the statement **S** is to be *executed*). Then some other statement may refer to the labeled statement simply by means of its label **L**.

Frequently it is necessary to consider a *compound statement*, i.e., a statement that itself consists of statements. Such compound statements are set between the words ***begin*** and ***end***; e.g., ***begin*** **S; S; S; S** ***end*** is a compound statement of four statements. A compound statement can also be labeled: for example, **; L : *begin* S; S; S; S *end*;**. The statements in a compound statement can themselves be labeled, as in

LC : *begin* L1 :S; L2 :S; L3 :S; L4 :S *end*

Here the compound statement label is **LC**, and the statement labels are **L1**, **L2**, **L3**, and **L4**.

The highest form in the hierarchy is the *block*, which is a collection of statements and compound statements, but which includes also the *associated declarations*. In a sense, a block is an entire code or program, just as a recipe is an entire description about how to prepare a particular type of food. A block can be labeled. For example, the form of a block (with **D**'s representing declarations) might be

LB : *begin* D; D; D; L1 :S; L2 :S; LC : *begin* S; S; S *end*; L3 :S *end*

where the block itself must be bracketed by the words ***begin*** and ***end***, and where **LB** is the block label, **LC** is a compound statement label, and **L1**, **L2**, and **L3** are statement labels. Of course blocks can contain blocks: for example,

LB1 : *begin* D; D; LB2 : *begin* D; D; S; S; S *end*; S; S; S *end*

where **LB1** and **LB2** are the labels of the two blocks involved. Since the block is essentially a complete entity within itself, every block automatically introduces a new level of nomenclature. In particular, *labels* of statements, of compound statements, and of blocks are considered to be *local to the block within which they occur*. By this is meant that a

labeled structure inside a block cannot be referred to by its label by a statement outside the block; and, conversely, a labeled structure outside a block cannot be referred to by its label by a statement inside the block. This is the same as saying that one cannot jump to a statement in the middle of a subroutine; rather the subroutine must be entered from the beginning. (See Grammar of Blocks, page 228.)

On the other hand, a *variable* or a *procedure identifier* can be declared (see *type declaration* below) to be *local* to (only) the block in which it occurs. Such a local variable or procedure identifier then has only the meaning intended in the block; the same variable or procedure identifier used outside the block will not refer to any value or procedure that occurs within this block. Variables or procedure identifiers *not* declared local to the block in which they occur will be understood to be *nonlocal* to the block, and hence to represent the same entity in the block level *immediately outside* the block as they do inside the block. Since a statement in a block may again itself be a block, the concepts of local and nonlocal to a block must be understood recursively. Thus a variable or procedure identifier which is nonlocal to block A may or may not be nonlocal to the block B which contains A as one of its components.

We turn now to a description of what each type of statement does, which is analogous to describing the meaning of operations in considering instructions. The descriptions of the five types of statements will be cast in a more formal organization than we have been using up to now. In the sequel for each type of statement we will first present its syntactical definition, then present a semantic description, and finally give a short grammatical discussion.

Assignment Statements. SYNTAX

⟨left part⟩ : : = ⟨variable⟩**:=**

⟨assignment statement⟩ : : = ⟨left part⟩ ⟨arithmetic expression⟩

| ⟨left part⟩ ⟨Boolean expression⟩

SEMANTICS. The assignment statement is probably the most important statement of ALGOL, although it is almost the simplest. As defined by this syntax, the assignment statement has the form

; VAR :=EXPR;

where **VAR** represents a variable (simple or subscripted) and **EXPR** can be an arithmetic or Boolean expression. It is a statement telling the computer to compute the value of the right-hand side and store that value (e.g., a number) in the storage location designated by the left part, i.e., to assign the value of the expression to the variable. The coloned

equal sign **:=** is used to indicate that the left part is to be the name (or address) of the value of the right-hand side. If the expression is in terms of numbers, such as **1.7↑2−4×3×(−0.31)**, then the indicated operations are performed. Thus

; Y:=1.7↑2−4×3×(−0.31);

tells the computer to assign **2.89+3.72**, or **6.61**, as the value of **Y**. A statement such as

; A:=3;

of course requires no computation and merely assigns the value of **3** to **A**.

GRAMMAR. If the Boolean or arithmetic expression of an assignment statement contains variables and procedure designators (i.e., functions), rather than simply numbers, then the computer obtains the values of the variables, which must have been previously assigned, and evaluates the functions, by means of appropriate subroutines. Thus to compute **(−B+*sqrt*(Y))/(2×A)**, where **Y** represents **B↑2−4×A×C** and **A** is **3**, **B** is **1.7**, and **C** is **−0.31**, we would write the following sequence of statements, which would be executed from left to right:

; A:=3; B:=1.7; C:=−0.31; Y:=B↑2−4×A×C;
root:=(−B+*sqrt*(Y))/(2×A);

In executing a sequence of assignment statements, the value of any variable is taken as the most recent value it attained during the execution of the sequence. If the new value of a variable is a function of the previous value of the variable, the variable's symbol appears on both sides of the statement; the most recent value is used on the right-hand side in computing the new value, which is then assigned to the left-hand symbol. Thus, if we desired to add **2** to **J**, where initially **J = 1**, we would write

; J:=1; J:=J+2;

where the second statement really means **J** (*new value*) = **J** (*previous value*) **+2**. Such increasing or updating of the value of a variable is an important method for handling indices during a loop or recursive routine.

***go to* Statements.** SYNTAX

⟨go to statement⟩ : : = ***go to*** ⟨designational expression⟩

SEMANTICS. A ***go to*** statement interrupts the normal sequence of execution of successive statements by explicitly defining the label of its successor to be the value of the designational expression of the statement. For example, if the label of the next desired statement, which is not in normal sequence, is **L**, then we would write simply

; *go to* L;

where **L** is a "designational expression" consisting only of a single label. As a more complicated example, consider the following:

; *go to if* B↑2>4×A×C *then* POS
else *if* B↑2<4×A×C *then* NEG *else* ZRO;

Here the alternative possible next statements to be executed are labeled **POS**, **NEG**, and **ZRO**, and which one is actually to be executed next depends on the current values of **A**, **B**, and **C**.

GRAMMAR. A ***go to*** statement *cannot* lead from outside a block into a block. This is because labels are inherently local, i.e., apply only to the block within which they occur. This becomes easy to see on recalling from above that essentially a block, like a subroutine, is operationally complete within itself.

Conditional Statements. SYNTAX

⟨unconditional statement⟩ : : = ⟨assignment statement⟩ | ⟨go to statement⟩ | ⟨procedure statement⟩
| ⟨for statement⟩ | ⟨compound statement⟩ | ⟨block⟩

⟨if statement⟩ : : = ⟨if clause⟩ ⟨unconditional statement⟩ | ⟨label⟩:⟨if statement⟩

⟨conditional statement⟩ : : = ⟨if statement⟩|⟨if statement⟩ ***else*** ⟨if statement⟩
|⟨if statement⟩ ***else*** ⟨statement⟩

⟨statement⟩ : : = ⟨conditional statement⟩ | ⟨unconditional statement⟩

SEMANTICS. The ***if*** statement is the basic ingredient of a conditional statement and is of the form

; *if* BEXP *then* S;

where **BEXP** is a Boolean expression (see page 211) of value **T** (true) or **F** (false) and **S** is an unconditional statement. For such a statement the computer would first determine whether **BEXP** is true or false: if **BEXP** is true, then **S** would be executed; otherwise **S** will be skipped, and the statement following the ***if*** statement will be executed next. Generally conditional statements will involve compounded ***if*** statements, and it is important to note that in the compound the ***if*** statements themselves may be labeled. For example, we could have a conditional statement of the form

; LCS:L1: *if* B1 *then* S1 *else* L2: *if* B2 *then* S2; S3;

where **LCS** is the label of the entire conditional statement, **L1** and **L2** are the labels of the first and second ***if*** statements, **B1** and **B2** are Boolean

expressions, **S1** and **S2** are the unconditional statements, and **S3** is the statement following the conditional statement. A second form of the conditional statement is allowed, which differs at the end: for example,

; LCS:L1; *if* B1 *then* S1 *else* L2: *if* B2 *then* S2 *else* S3; S4;

where again **LCS**, **L1**, and **L2** are, respectively, the labels of the entire conditional statement and each *if* statement, **B1** and **B2** are the Boolean expressions, **S1** and **S2** are unconditional statements, and **S3** and **S4** are statements that are not necessarily unconditional. These two forms are illustrated in Fig. 6-4.

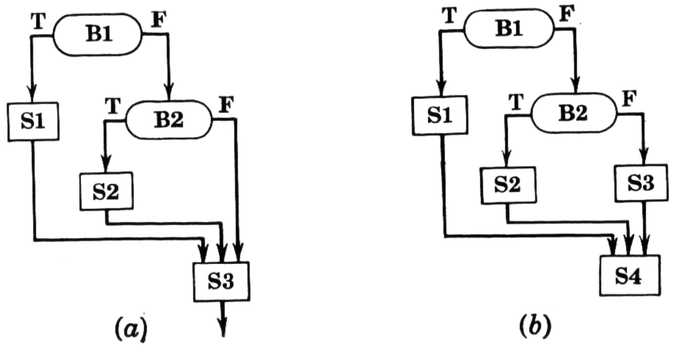
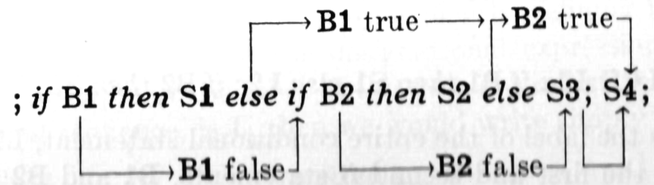


FIG. 6-4. (*a*) Flow diagram of the first conditional form; (*b*) flow diagram of the second conditional form.

The execution of a conditional statement progresses from left to right, with each successive Boolean expression of the *if* statements being evaluated; only after the first *true* Boolean expression is found is the unconditional statement associated with that *if* statement executed. (When executing a conditional statement, care must be taken to distinguish between the *if* as used in the statement and the *if* that may appear in a component expression of the statement!) If this unconditional statement is a *go to* statement, then it will define the label of the next statement to be executed. If it is not a *go to* statement, or if none of the Boolean expressions is true, then the statement following the entire conditional statement is executed next (see Fig. 6-4). The following diagram illustrates this situation for the second of the above forms, involving no *go to* statements:



GRAMMAR. Frequently the *go to* statement is used in conjunction with the conditional statement. As an example of such use, consider the problem of finding the largest of three given numbers **U**, **V**, and **W**. The sequence of statements, according to Fig. 2-3 (page 70), will be

; *if* U>V *then go to* UW *else if* V>W *then go to* VLN;
WLN:LNC:=W; *go to* T; VLN:LNC:=V; *go to* T;
UW: *if* U>W *then* LNC:=U *else go to* WLN; T: *stop*;

Here the largest number is to be assigned as the value of **LNC**. Four statements were labeled to be referenced by the *go to* statements, namely, **WLN**, **VLN**, **UW**, and **T**. Two conditional statements were used. This is somewhat reminiscent of a one-address code.

for Statement. SYNTAX

⟨tally condition⟩ ::= ⟨arithmetic expression⟩
| ⟨arithmetic expression⟩ ***step*** ⟨arithmetic expression⟩ ***until*** ⟨arithmetic expression⟩
| ⟨arithmetic expression⟩ ***while*** ⟨Boolean expression⟩

⟨for list⟩ ::= ⟨tally condition⟩|⟨for list⟩,⟨tally condition⟩

⟨for statement⟩ ::= ***for*** ⟨variable⟩:=⟨for list⟩ ***do*** ⟨statement⟩

SEMANTICS. The automatic compilation of loops, or recursive codes, is accomplished by the *for* statement. The *for* statement causes the statement following the *do* (which may be a compound statement) to be executed repeatedly, where the value of some variable is changed or updated for each repetition. The *for* statement automatically senses the tally and successively evaluates the conditions stated by the "for list" of the *for* statement until every one of the tally conditions of the "for list" is met. There are three kinds of tally conditions: (1) those that simply specify a particular list of successive conditions which are to be exhaustively met; (2) those that count the number of iterations by *stepping* up some parameter *until* the value of the parameter is greater than some tally constant; and (3) those that successively utilize the result of each iteration to evaluate a relation, with iteration continuing only *while* the given relational test holds. The three types of "for elements" of the syntax are designed to meet these three types of tallies, as will now be illustrated.

The form of the *for* statement is

; *for* VAR:=TC1,TC2, . . . ,TCN *do* S;

where **TC1**, **TC2**, . . . , **TCN** are the "tally conditions" which comprise the "for list" and **S** is a statement. The *for* statement successively

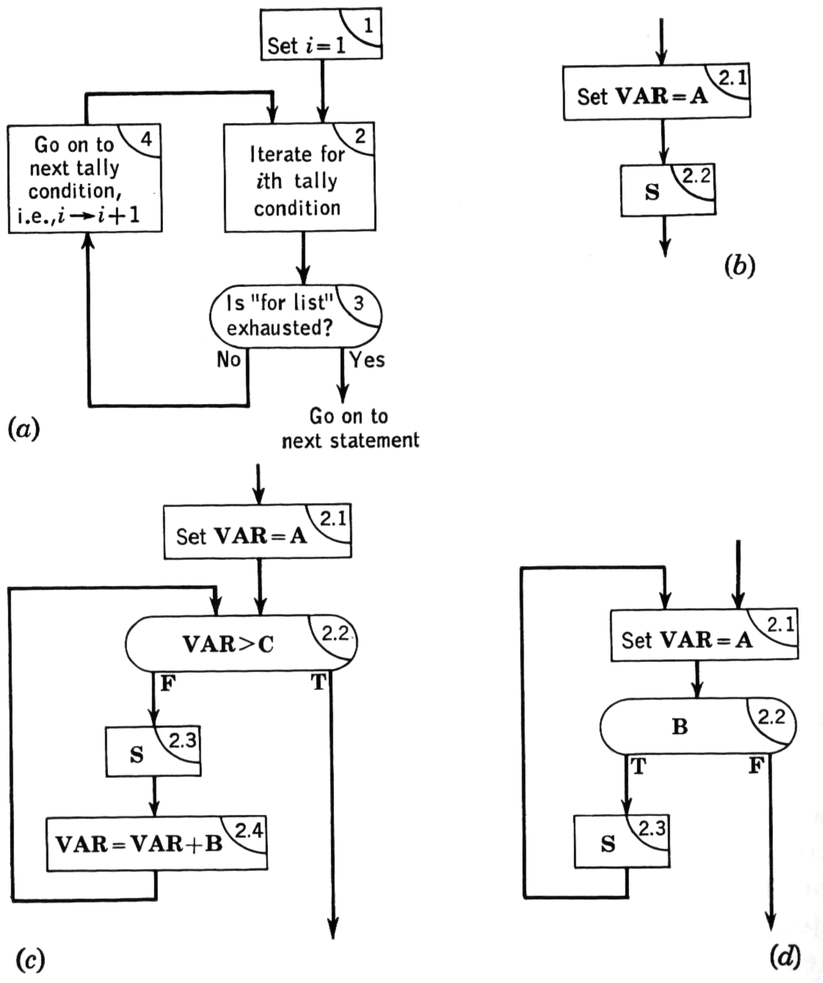


FIG. 6-5. (*a*) Loop illustrating successive evaluation of tally conditions; (*b*) the first type of tally condition of the form ***for*** **VAR:=** . . . , **A,** . . . ***do*** **S** is simple substitution of **A**; (*c*) the second type of tally condition of the form

for **VAR:=** . . . , **A** ***step*** **B** ***until*** **C,** . . . ***do*** **S**

involves a loop with successive updating of **VAR** by **B** on each iteration; (*d*) the third type of tally condition of the form ***for*** **VAR:=** . . . , **A** ***while*** **B,** . . . ***do*** **S** involves a loop where the evaluation of **S** itself updates **A** on each iteration.

evaluates **S** for **VAR**, taking on the successive values as directed by the "for list" (see Fig. 6-5*a*). The first type of tally condition is of the form

; ***for*** **VAR:=** . . . **,A,** . . . ***do*** **S;**

(see Fig. 6-5*b*), where **A** is an arithmetic expression that is simply substituted for **VAR** when **S** is executed. Several of these can be used in a for list, as in the evaluation of **Y = a(Z+b(Z+cZ))**:

;Y:=0; ***for*** **VAR:=c,b,a** ***do*** **Y:=VAR×(Z+Y);**

Here we initially set **Y** at 0 and then successively add **Z** and multiply by coefficients **c**, **b**, and **a**. These coefficients could themselves be arithmetic expressions, of course.

The form of the *for* statement using the second type of tally condition is

; ***for*** **VAR:=** . . . **,A** ***step*** **B** ***until*** **C,** . . . ***do*** **S;**

where **A**, **B**, and **C** are arithmetic expressions and **S** is a statement. Here the initial value of the variable **VAR** is set equal to the value of **A**; the statement **S** is then executed. Next a new value of **A** is computed as the sum of the present value of **A** and the value of **B**; if this new value is less than or equal to the value of **C**, **S** is executed again, using the new value of **A**. When the new value of **A** becomes greater than that of **C**, then the statement following the ***for*** statement is executed next. We can write this explanation in terms of ALGOL as follows (see Fig. 6-5*c*):

;VAR:=A; L: ***if*** **VAR>C** ***then go to*** **EXIT; S;**
VAR:=VAR+B; ***go to*** **L; EXIT;**

where **EXIT** is the next tally condition, or the next statement if the "for list" is exhausted. So far we have considered (the value of) **B** > 0; if **B** < 0, then we define the operation of the ***for*** statement as follows:

;VAR:=A; L: ***if*** **VAR<C** ***then go to*** **EXIT; S;**
VAR:=VAR+B; ***go to*** **L; EXIT;**

As an example of the ***step-until*** type of ***for*** statement, consider again the problem of finding the largest of three given (positive) numbers **N[1]**, **N[2]**, and **N[3]**:

;LNC:=0; ***for*** **J:=1** ***step*** **1** ***until*** **3** ***do***
if **LNC<N[J]** ***then*** **LNC:=N[J];** ***stop*****;**

Here the "for list" has a single tally condition; we initially set the contents of **LNC** at **0**; then we compare successively each of the three numbers with the contents of **LNC**, assigning the number to **LNC** if it is larger than **LNC**; finally, the value of **LNC** will be the largest of the three numbers. The "arithmetic" expressions **A**, **B**, and **C** turned out for this example to be just the numbers **1**, **1**, and **3**, respectively: such simple arithmetic expressions will frequently occur. Note that the use of the ***if*** statement interprets the "next statement" to be the ***if*** statement again with the index **J** changed; this is a common use of the ***for-if*** statement combination.

The index **VAR** of a ***for*** statement need not appear explicitly in the statement **S**. For example, consider the computation of $\sum_{i=1}^{5} M_i$, where

$M_{j+1} = 3M_j^2 - 9M_{j-1}^2$, $M_0 = 0$, and $M_{-1} = 1$. The code is

```
; SUM:=0; Ma:=1; Mb:=0; for J:=1 step 1 until 5 do
      begin Mc:=3×Mb↑2−9×Ma↑2; SUM:=SUM+Mc;
                  Ma:=Mb; Mb:=Mc end; stop;
```

Each time around the loop, **J** is increased by **1**; then the process ends after five iterations, as desired, when **J** becomes **5**. Note that **a**, **b**, and **c** really correspond to $j - 1$, j, and $j + 1$, respectively.

The form of the *for* statement using the third type of tally condition is

```
; for VAR:= . . . ,A while B, . . . do S;
```

where **A** is an arithmetic expression, **B** is a Boolean expression, and **S** is a statement. Here the initial value of the **VAR** must have been set before the occurrence of this tally condition, by a previous tally condition or a statement before the *for* statement. The new, updated value of **VAR** is computed by **A** for each iteration. The statement **S** often computes the value of some variable that is involved in the Boolean expression **B**; at other times, **B** may involve the variable **VAR**. Then the sequence of computations carried out by the *for* statement can be summarized (see Fig. 6-5*d*) as

```
; L:VAR:=A; if ¬B then go to EXIT; S; go to L; EXIT;
```

Here the new value of the variable is determined first; then **B** is evaluated: if **B** is true, then ¬**B** is false and the statement **S** is executed; otherwise **B** is false, ¬**B** is true, and the exit is to be taken next.

As an example, consider the computation of

$$\sin \theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \cdots$$

while

$$\frac{\theta^{2n+1}}{(2n+1)!} > \varepsilon$$

The code is as follows:

```
; TERM:=θ; SUM:=θ;
   for J:=1,J+1 while TERM>ε do
      begin TERM:=(−1×TERM×θ↑2)/((2×J+1)×2×J);
                  SUM:=SUM+TERM end; VSIN:=SUM; stop;
```

Here we multiply the previous term by $-\theta^2/((2J + 1)2J)$ to obtain the present term; the partial sum is then formed—that is, **S** is a compound statement. This process is repeated as long as **TERM>ε**. Note that we initially do the computation for **J** = **1**, and then we go into the *while* type of tally condition with **J** = **J+1**. Such initialization by a tally

condition of the first type is typical of the use of the *while* type of tally condition.

All three types of tally conditions can be used in the same *for* statement. For example,

$$\sum_{n=1}^{9} n^2$$

would be formed by

```
; SUM:=0; for n=1,2,3,4 step 1 until 6,n+1 while n≦9
                  do SUM:=SUM+n↑2;
```

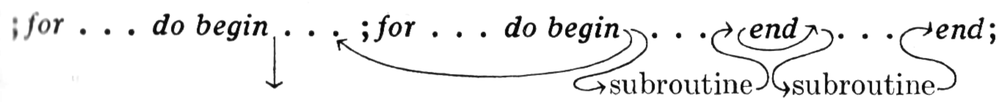
GRAMMAR. Loops can contain loops; in other words, *for* statements can contain *for* statements. For example, consider the code for

$$\sum_{i=1}^{5} \left[\sum_{j=1}^{3} (i+1)(j+2) \right]$$

namely,

```
; SUM:=0; for i:=1 step 1 until 5 do
begin for j:=1 step 1 until 3 do SUM:=SUM + (i+1)×(j+2) end; stop;
```

A general rule to apply is that no jump may be made into the range of a *for* statement (i.e., into the "*do* statement" **S**, which may be a compound statement) except during the use of subroutines not affecting the *for* indices. The following diagram indicates valid jumps (i.e., *if* followed by *go to* statements, or just *go to* statements):



If the exit was made through a *go to* statement (as part of the compound statement **S**), the value of the index, or *controlled variable* (i.e., **VAR** in the above forms), will be its final value reached just before the exit. If the exit was due to the completion of the index or controlled-variable range, then the value of the controlled variable is *undefined* after the exit.

6-5. Type, *Array*, and *Switch* Declarations

Need for Declarations. In our chocolate fudge illustration of Sec. 6-1, we observed that, before the actual procedure of the recipe was given, a list of ingredients and certain special notes were needed. In our discussions in Secs. 6-2 through 6-4, we have defined the basic expressions and statements of our language, in terms of which we can write the actual code, which are analogous to the procedure of the recipe. We turn now to the structures in terms of which we write the declarations of our lan-

guage, which are analogous to the list of ingredients and the notes of our recipe. In this section we shall consider those declarations that describe the nature of the variables involved (analogous to the ingredients list); and in the next section we shall describe the procedure declaration, or subroutine (analogous to the notes of our recipe).

Type and ***Array*** *Declarations.* In order that the computer will know what kinds of operations may be performed on the variables involved in a code, and also that it will have a record of the names of these variables, the variables are declared as to the *type* of value they represent. The types of variables are (1) ***real***, meaning such variables may assume positive, zero, or negative values; (2) ***integer***, meaning such variables may assume only positive and negative *integral* values, including zero; and (3) ***Boolean***, meaning such variables may assume only the truth values **T** and **F**, for *true* and *false* respectively. The syntax of a *type declaration* would therefore be

⟨type⟩ : : = ***real***|***integer***|***Boolean***
⟨variable list⟩ : : = ⟨variable⟩|⟨variable list⟩,⟨variable⟩
⟨type declaration⟩ : : = ⟨type⟩⟨variable list⟩

For example, for our illustration of finding the quadratic root, we would declare

real **A,B,C**

whereas in the double summation illustration above we would declare

integer **i,j**

In addition to those specifying the type, there are specification declarations that describe variables (usually subscripted) which represent *arrays*, such as vectors or matrices. The ***array*** *declaration* tells the computer how many elements there are in the vector or matrix, so that the appropriate memory space may be reserved. The form of an ***array*** declaration is

array **VAR[Li :Ui,Lj :Uj]**

where for each index of the variable **VAR** the *lower* and *upper bounds* are given. Thus for the matrix

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

we would write ***array*** **MATRIX[1 :3,1 :2]**. For a vector **V** with five components (V_1, V_2, V_3, V_4, V_5), we would write ***array*** **V[1 :5]**, since there is only a single index involved. The notation extends to any number of indices, and if more than one array is being declared at the same time,

the list is separated by commas; e.g.,

array **MATRIX[1 :3,1 :2],V[1 :5]**

For greater generality, the upper and lower bounds may themselves be arithmetic expressions. Also, in order that the "type" of the array elements be known, the type is usually specified before the word *array*. Thus we might have

integer array **V[r—s :r+s]**

where there are 2**s** + 1 elements in the vector, each of which is an integer.

Switch *Declarations.* The ***switch*** *declaration* acts as a *variable connector* by specifying the *labels* of alternative statements to which a ***go to*** statement may jump. The syntax of a ***switch*** declaration is as follows:

⟨switch list⟩ : : = ⟨designational expression⟩ | ⟨switch list⟩, ⟨designational expression⟩

⟨switch declaration⟩ : : = ***switch*** ⟨variable⟩ **:=** ⟨switch list⟩

The semantics of the ***switch*** declaration is as follows: The switch list is in effect an *ordered list of labels*, since it will be recalled that a designational expression is itself a label or has a label as its value. Thus its form is

switch **NAME :=LA,LB,LC**

where **NAME** is the name of the switch itself and **LA,LB,LC** is the list of labels. The switch, or variable connector, is used by a ***go to*** statement of the syntactical form

go to ⟨switch designator⟩

where for "switch designator" we have

⟨switch designator⟩ : : = ⟨variable⟩[⟨integer⟩]

Thus the switch designator is of the form

NAME[N]

where **NAME** is the name of the switch corresponding to the switch declaration and **N** is an *integer* telling which label of the switch declaration to use. Thus if **LA**, the first label, is to be used, then **N** would be set to **1**, and we would write ***go to*** **NAME[1]**; if the second label, **LB**, is to be used, then **N** would be set to **2**; and so forth. For example, consider the problem of forming

$$\begin{aligned} A &= X^2 + Y && \text{if } \mathbf{N} = \mathbf{1} \\ A &= X + Y^2 && \text{if } \mathbf{N} = \mathbf{2} \\ A &= 0 && \text{if } \mathbf{N} = \mathbf{3} \end{aligned}$$

We would form the required **N**, and then by means of the switch we would jump to the proper assignment statement:

```
; switch SW:=P,Q,R; J:=N; go to SW[J];
        P:A:=X↑2+Y; go to T; Q:A:=X+Y↑2; go to T
                                          R:A:=0; T:stop;
```

We could have used a more general description of the "switch designator" by utilizing an integral-valued arithmetic expression in the square brackets rather than only an integer. Similarly, we could generalize the switch list to itself include switch lists by extending the syntax of the designational expression to include switch lists (see Exercises 6-43 and 6-44).

Grammar of Blocks: Local and Nonlocal Variables. Recall from above that a block is essentially a compound statement that includes some declarations at its beginning. A declaration for a variable specifies the nature of that variable for one block only; i.e., the declared variable is *local* to the block in which it is declared. Outside that block, the variable can be used for other purposes but can never refer to its use within the block in question. Hence a variable used inside a block but not declared in it, i.e., not included in a *type* or ***array*** declaration in the block—such a variable is *nonlocal* with respect to the block and will represent the same entity immediately outside the block as well as within it. Of course *every* variable must be declared in some block. Then a variable represents the same entity for all blocks which are subcomponents of the block in which it was declared. Labels of statements are *always* local to the particular block in which they occur. Sometimes the term *scope* is used for the range of statements for which a variable has the same meaning. Then the scope of a variable is the set of statements within the block for which the specifying declaration for the variable was made—and this includes all "subblocks" within this block. When a statement is entered, the value of any variable is always assumed to be its most recent value, in the "local" sense.

6-6. Procedure Statements and Declarations

Need. The statements discussed above—the assignment, ***go to***, conditional, and ***for*** statements—enable the evaluation of arithmetic and Boolean expressions and also present a certain amount of "decision-making" and "jumping" capability. These statements are sufficient to handle any algebraic computation. Clearly, however, there will be a large variety of other entities which, though they cannot be predicted in advance, will be suggested by the nature of specific problems as they occur. Hence room must be made in our language for the incorporation of special computational entities as desired—or, in more familiar words, there must be a method in our language for the use of *subroutines*. Of course, these computational entities will themselves be expressed in terms of the four basic statements. This requirement results in the need for *procedure statements* and *procedure declarations*. A procedure statement tells the computer to execute a procedure declaration, which latter is the subroutine itself.

Procedure Statements. SYNTAX

⟨actual parameter⟩ : : = ⟨variable⟩|⟨expression⟩|⟨function designator⟩

⟨actual-parameter list⟩ : : = ⟨actual parameter⟩ | ⟨actual-parameter list⟩, ⟨actual parameter⟩

⟨actual-parameter part⟩ : : = ⟨empty⟩ | (⟨actual-parameter list⟩)

⟨procedure statement⟩ : : = ⟨procedure identifier⟩ ⟨actual-parameter part⟩

SEMANTICS. A procedure statement tells the computer to execute a procedure declaration or subroutine which appears in the declaration of a program. The "procedure identifier" constitutes the name of the subroutine (just as it constituted the name of the function as described above), and the "actual-parameter list" in the parentheses is a list of both the *dependent* and *independent* variables involved. The form of the procedure statement is

; PROCID(AP1,AP2,AP3);

where **PROCID** is the procedure identifier and, say, **AP1** is the dependent variable and **AP2** and **AP3** are the independent variables. Frequently there may be more than one dependent variable computed by a procedure statement. For example,

; TRIG(θ,ε,EXPO,SIN,COS);

might be the procedure that tells the computer to execute a subroutine denoted by **TRIG** that has as its independent variables an angle **θ** and an accuracy **ε** and computes as dependent variables **EXPO**, whose value is e^{θ}; **SIN**, whose value is sin θ; and also **COS**, whose value is cos θ. Of course, to find out just what the meaning of the dependent and independent variables actually is, one must refer to a description of the corresponding procedure declaration or subroutine.

GRAMMAR. The procedure statement does not *necessarily* use the same "actual parameters" in the actual-parameter part as the procedure declaration itself does. For example,

; TRIG(ω×t,ε,U,V,W);

would also invoke (call in) the **TRIG** subroutine: the *order of the "actual parameters"* identifies the actual correspondence with the dependent and independent variables of the subroutine. Thus in our example **ω×t** corresponds to **θ**, **ε** to **ε**, **U** to **EXPO**, **V** to **SIN**, and **W** to **COS**. Suppose we desired to use this subroutine to compute

$$e^{-\omega t}[A \sin(\omega t) + B \cos(\omega t)]$$

The procedure statement in the actual code would be

. . . ; TRIG(ω×t,ε,U,V,W); Z:=(A×V+B×W)/U;

Here **U**, **V**, and **W** would be computed first by means of the **TRIG** subroutine, and then their values would be used to compute **Z** (it is of course assumed that the values of **ω**, **t**, and **ε** were computed by previous statements in the program, represented by . . .).

Procedure Declarations. Finally, we come to the procedure declaration itself. Its syntax is as follows:

⟨formal parameter⟩ : : = ⟨simple variable⟩

⟨formal-parameter list⟩ : : = ⟨formal parameter⟩ | ⟨formal-parameter list⟩, ⟨formal parameter⟩

⟨formal-parameter part⟩ : : = ⟨empty⟩ | (⟨formal-parameter list⟩)

⟨variable list⟩ : : = ⟨variable⟩|⟨variable list⟩, ⟨variable⟩

⟨value part⟩ : : = ⟨empty⟩|⟨variable list⟩;

⟨type⟩ : : = ***real*|*integer*|*Boolean***

⟨specifier⟩ : : = ⟨type⟩|***switch***|⟨type⟩***array***|⟨type⟩ ***procedure*** ⟨function designator⟩

⟨specification part⟩ : : = ⟨empty⟩|⟨specifier⟩⟨variable list⟩;
| ⟨specification part⟩ ⟨specifier⟩⟨variable list⟩;

⟨procedure heading⟩ : : = ⟨procedure identifier⟩ ⟨formal-parameter part⟩; ⟨value part⟩ ⟨specification part⟩

⟨procedure body⟩ : : = ⟨statement⟩

⟨procedure declaration⟩ : : = ***procedure*** ⟨procedure heading⟩ ⟨procedure body⟩

The semantics of a procedure declaration are as follows: The word ***procedure*** indicates that what follows is a subroutine; the "procedure heading" is the name of the subroutine and is followed by the list of "formal parameters" (the independent and dependent variables) in parentheses, together with certain specifications, concerning the variables, which enter into the subroutine. Note that the "procedure body," i.e., the subroutine itself, is defined as a *single statement*, and hence must be written as a single compound statement, in the usual ***begin*** . . . ***end*** notation for such statements. Thus the form of a procedure declaration will be

procedure **NAME(V1,V2, . . . ,VN);** ⟨value part⟩ ⟨specification part⟩ ***begin*** . . . ***end*;**

(***procedure*** **NAME(V1,V2, . . . ,VN);** ⟨value part⟩ ⟨specification part⟩) = procedure-declaration heading

(***begin*** . . . ***end*;**) = procedure body, i.e., main part of subroutine

All "formal parameters," i.e., the independent and dependent variables listed in the formal-parameter part (for example, **V1**, **V2**, . . . , **VN** of our form), *must be declared as to type* in the "specification part." If one of the formal parameters is an array, it must be declared as an ***array,*** and also as to "type" of elements of which the array is composed. Any switch used in a procedure body may be declared in the "specification part," but as was seen above, switch declarations may also be used in the body itself. Note that the procedure may involve some function: in the formal-parameter part, the function is indicated only by its "procedure identifier," or variable name; in the specification part, however, the type and the function designator are listed. We shall get to the meaning of the "value part" in a moment.

The effect of a procedure *statement* is to replace the statement by the procedure *declaration* and to execute the "statement" that comprises the subroutine of the declaration. Recall from above that the formal parameters of the procedure declaration are associated with the actual parameters of the procedure statement in a one-to-one ordered correspondence, the first formal parameter corresponding to the first actual parameter, the second formal parameter corresponding to the second actual parameter, and so forth, from left to right. When the procedure declaration replaces the procedure statement, those formal parameters that are *independent variables* of the procedure must take on the most recent values of the corresponding actual parameters. No problem results when each actual parameter is a single variable, for the corresponding formal parameters will then be replaced by the most recent values of the actual parameters in the body of the procedure. But consider the case where the actual parameter is itself an expression involving some arithmetic or logical operations. Then there are two possible courses of action: the first is

to have the expression evaluated, and then have this value replace the corresponding formal parameter in the procedure body; the second is to replace the formal parameter in the procedure body by the corresponding actual parameter expression enclosed in parentheses, leaving its evaluation to be accomplished during the execution of the procedure. The first course of action is best when the formal parameter occurs frequently in the procedure body, for then the corresponding actual parameter expression need only be evaluated once. Those formal parameters for which this first course of action is desired are indicated by listing them in the "value part" of the procedure heading.

Examples of Procedure Declarations. As a first example, consider the procedure declaration corresponding to the procedure statement

; TRIG(ω×t,ε,U,V,W);

used above in the evaluation of $e^{-\omega t}[A \sin(\omega t) + B \cos(\omega t)]$. The procedure computes

$$e^{\theta} = 1 + \theta + \frac{\theta^2}{2!} + \frac{\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{\theta^5}{5!} + \cdots$$

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \cdots$$

$$\cos\theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \cdots$$

To evaluate these three functions in the same procedure, we could first form $\theta^m/m!$ and then add this (with the appropriate sign) to the respective partial sums as called for by the series. We use a switch declaration for the alternation of sign for the sine and cosine. The procedure declaration would be

```
procedure TRIG(θ,ε,EXPO,SIN,COS); value θ,ε;
  real θ,ε,EXPO,SIN,COS; switch SAC:=S,C;
  begin real SUME,SUMC,SUMS,TERM; integer I,J;
  TERM:=θ; SUME:=1+θ; SUMS:=θ; SUMC:=1; J:=2; I:=2;
  B:TERM:=TERM×θ/J;
  if TERM>ε then SUME:=SUME+TERM else go to R; go to SAC[I];
  C:SUMC:=SUMC+(−1)↑(J/2)×TERM; I:=1; go to T;
  S:SUMS:=SUMS+(−1)↑((J−1)/2)×TERM; I:=2; go to T;
  T:J:=J+1; go to B;
  R:EXPO:=SUME; SIN:=SUMS; COS:=SUMC end
```

Note that the inputs to the subroutine are **θ** and **ε**, in that order; the outputs are **EXPO**, **SIN**, and **COS**, in that order. The switch is reset to go through the alternate sin (or cos) loop at the end of each loop. Finally,

the partial sums **SUME**, **SUMS**, and **SUMC** are assigned to their corresponding variables.

As another example of a procedure declaration, consider matrix multiplication, where it is desired to compute

$$C_{ik} = \sum_j A_{ij}B_{jk} \qquad i = 1, \ldots, I; j = 1, \ldots, J; k = 1, \ldots, K$$

The procedure declaration becomes

```
procedure MATMULT(A[i,j],B[j,k],C[i,k]); value I,J,K;
  integer I,J,K,i,j,k; real array A[1:I,1:J],B[1:J,1:K],C[1:I,1:K];
  begin for i:=1 step 1 until I do
  begin for k:=1 step 1 until K do
  begin C[i,k]:=0; for j:=1 step 1 until J do
  C[i,k]:=C[i,k]+A[i,j]×B[j,k] end end end
```

As a final example consider integration by means of the trapezoidal rule, namely,

$$A = \int_a^b f(x)\,dx = \frac{h}{2}[f(a) + 2f(a+h) + 2f(a+2h) + \cdots + f(b)]$$

Here we shall utilize what is called a ***comment*** declaration, which is simply a way of inserting into the procedure descriptions that are ignored by the computer but are of some use to the programmer when reading the procedure declaration.

```
procedure TRAP(FCT,a,b,ε,V,A); value a,b,ε,V; real a,b,ε,V,A
  real procedure FCT(x); comment a and b are the min and max, resp.,
  of the points defining the interval of integration. FCT(x) is the function
  to be integrated, ε is the permissible difference between two successive
  trapezoidal sums, and V is greater than the max absolute value
  of FCT(x) on (a,b);
  begin real Ibar,h,J,S,I; integer n,i,k;
  h:=b−a; Ibar:=V×h; J:=1/2×(FCT(a)+FCT(b)); I:=h×J
  S:=0; n:=1; i:=0; for i:=i+1 while abs(I−Ibar)>ε do
  begin h:=h/2; Ibar:=I; for k:=1 step 1 until n do
  S:=S+FCT(a+h×(2k−1));
  I:=h×(J+S); n:=2↑i end;
  A:=I end
```

Note that for the ith iteration we add 2^i more points to **S**, namely, the new points between the previous points. Then

h×((1/2)×(FCT(a)+FCT(b))+S)

becomes the new approximation to the integral. Note that the standard

function ***abs*(y)**, the absolute value of y, need *not* be declared. (See also Exercise 6-50.)

Functions and Procedure Statements. The procedure statement has, then, the same effect as a function. The differences, however, are as follows: First, the "function designator" can stand only for a *single-valued* function, whereas the procedure statement may call for a multi-valued result. Second, the independent-variable list, analogous to the "actual-parameter list," contains only *independent* variables, whereas, for the procedure statement, both dependent and independent variables are included. For these reasons, the function designator can be included within an arithmetic expression, while clearly the procedure statement could not be so included. In fact, it is for the same reasons that the procedure statement has to be a complete statement by itself. Of course, the same effect of a function designator can be obtained with a procedure statement. For example, suppose that ***sqrt*(Q,Y)** is the procedure statement that computed $Q = \sqrt{Y}$. Then our quadratic root would be computed by

```
; A:=3; B:=1.7; C:=-0.31; Y:=B↑2-4×A×C;
                              sqrt(Q,Y); root:=(-B+Q)/(2×A);
```

where ***sqrt*(Q,Y)** would call in the square-root subroutine.

The use of a procedure declaration for a function is similar to the use of an ordinary declaration in that the actual parameters of the function designator correspond in order to the formal parameters of the function procedure. For example, suppose **QFT(u,v)** is the function designator for

$$f(u,v) = au^2 + buv + cv^2$$

To see the use of the function in a code, consider the evaluation of

$$Z = Af(r^2,s^2) + Bf(r^3,s^3)$$

After declaring the function in a procedure declaration (see below), we would write

```
Z:=A×QFT(r↑2,s↑2)+B×QFT(r↑3,s↑3)
```

The procedure declaration for a function is the same as that for an ordinary procedure statement except that the *type* must precede the word ***procedure*** and that a dependent variable never appears. Instead, within the procedure body a value must be assigned to the function identifier, i.e., to the name of the function. For our **QFT(u,v)** we would have, for $a = 3.8$, $b = 6.2$, and $c = 1.1$,

```
real procedure QFT(u,v); value u,v; real u,v; begin
    a:=3.8; b:=6.2; c:=1.1; QFT:=a×u↑2+b×u×v+c×v↑2 end
```


As another example of a function procedure consider the evaluation of $\sin \theta$ by a series:

```
real procedure sin(θ,ε); value θ,ε; real θ,ε; begin
  real TERM,SUM; integer J; TERM:=θ; SUM:=θ;
  J:=0; for J:=J+1 while TERM>ε do
  begin TERM:=(-1×TERM×θ↑2)/((2×J+1)×2×J);
  SUM:=SUM+TERM end; sin:=SUM end
```

To use this function procedure consider the computation of

$$Z = 3 \sin^2 (u - v, \varepsilon) + \sqrt{1 - \sin^2 (u^2 - v^2, \varepsilon)}$$

for $\varepsilon = 1/n$. After declaring ***sin*(θ,ε)** we would write

```
; Z:=3×(sin(u-v,1/n))↑2 + sqrt(1-(sin(u↑2-v↑2,1/n))↑2);
```

Since we have declared the ***value*** of θ and ε, the computer would first compute **u-v** and **1/n** for the first ***sin*** evaluation and **u↑2-v↑2** and **1/n** for the second ***sin*** evaluation before entering into the procedure itself either time.

Procedure declarations can utilize themselves (by means of the corresponding procedure statements) recursively. For example, consider

```
real procedure FACTORIAL(n); value n; integer n;
    begin FACTORIAL:= if n=0 then 1 else n×FACTORIAL(n-1) end
```

6-7. Exercises

6-1. List at least 10 more characters used in English in addition to the 77 mentioned in Sec. 6-1. (HINT: Look at a typewriter keyboard.)

6-2. What difficulties are involved in defining a "well-formed," intelligible word?

6-3. Define a "well-formed" number. (HINT: See Sec. 9-11 on significant figures.)

6-4. For written English, define *phrase*, *clause*, and *paragraph*.

6-5. Define a "well-formed" algebraic equation.

6-6. Identify the following as an unsigned integer, or a decimal fraction, or a decimal number, or a number, or some combination of these:

(*a*) .36.	(*b*) . 28.	(*c*) −3.65.
(*d*) .028.	(*e*) 0100.01.	(*f*) 000.000.

6-7. Discuss the semantics of "numbers" (i.e., the ordinal and cardinal meanings).

6-8. Discuss in detail the semantics of a *verb*.

6-9. Use the meta-metalanguage symbolism to define a "well-formed" number (see Exercise 6-3).

6-10. Extend the metalanguage, or symbolic-syntax, definition of a number given in Sec. 6-1 to include the exponent part of the form $_{10}\pm$**aa . . . a**.

6-11. Write a symbolic syntactical definition of a subscripted variable. (HINT: Use the definition of a *function designator* of Sec. 6-2 as a guide.)

Calculate the truth values of the following three Boolean expressions (for real numerical values):

6-12. $|a| + |b| \geq \sqrt{a^2 + b^2} \wedge p + q > p - q$.

6-13. $p^2 > q^2 \supset p^2 + q^2 > p^2 - q^2 \equiv p > q \vee p < q \vee p = q$.

6-14. For $a = 3$, $b = 6$, $c = -4$, and $d = 8$ evaluate

$$\left(a + \frac{b}{c} > a^2 + bc \equiv a^3 - d^2 \leq abcd\right) \vee b^2 - 4ac > b - 4a^2c \wedge a < d$$

6-15. Evaluate the expression of Exercise 6-14 with $a = 8$, $b = -4$, $c = 6$, and $d = 3$.

6-16. In the expression

u×w+v≥y∧u×v<z∨(a∨b≡¬c⊃d)

if the parentheses were omitted, how would the order of evaluation be affected?

6-17. Identify the line of the syntactical definitions which is utilized in each stage of the development of the simple Boolean expression given on page 210.

6-18. Diagram the expression of Exercise 6-14 as illustrated on page 210.

Evaluate the following three Boolean expressions (where **a**, **c**, **f** are false and **b**, **d**, **g** are true):

6-19. ***if if if*** **a** ***then*** **b** ***else*** **c** ***then*** **d** ***else*** **f** ***then*** **g** ***else*** **sin2+.05>tan2**

6-20. ***if*** **a** ***then*** **¬b** ***else if*** **c** ***then*** **d** ***else if*** **f** ***then*** **g** ***else*** **sin2+.05>tan2**

6-21. ***if if*** **a∨b∧c≡f** ***then*** **a⊃b≡c∧d** ***else*** **T** ***then*** **F** ***else*** **T**

6-22. Express the Boolean expression

if **a** ***then*** **b** ***else*** **c**

in terms of the Boolean operations only. (HINT: Write out the truth table for the expression.)

6-23. Express the Boolean expression

if if **a** ***then*** **b** ***else*** **c** ***then*** **d** ***else*** **a**

in terms of the Boolean operations only.

For **a = 3**, **b = 4**, **c = 5**, evaluate the following five arithmetic expressions:

6-24. **a−b/c×6/(b−c)↑2×a/b**

6-25. **b/c/b×a↑3×c↑a/b**

6-26. ***if*** **a−b/c↑2≥b−c/a↑c** ***then*** **a−b/c×6/(b−c)↑2×a/b** ***else*** **b/c/b×a↑3×c↑a/b**

6-27. ***if*** **a>b** ***then*** **c** ***else if*** **b>c** ***then*** **a** ***else if*** **c>a** ***then*** **b** ***else*** **c**

6-28. ***if*** **b>c** ***then*** **a** ***else if*** **c>a** ***then*** **b** ***else if*** **a>b** ***then*** **c** ***else*** **a**

6-29. Diagram the expression for ***root*** shown on page 205, as illustrated on page 213; compare the diagram with that for ***root*** on page 213.

Perform the indicated statements and find the value of **Z**:

6-30. **A:=3; B:=1.7; C:=−0.31; Y:=** ***if*** **B↑2−4AC≤0** ***then*** ***sqrt***(**A+B**) ***else*** ***sqrt***(**B−C**)**; Y:=Y+10; Z:=Y**

6-31. **A:=3; B:=1.7; A:=A+B; B:=B↑2+2×A; Z:=A+B**

6-32. **A:=3; L:A:=A+2;** ***go to if*** **10×A−A↑2≥0** ***then*** **L** ***else*** **M; M:Z:=A↑2−15**

6-33. **A:=U; B:=V; C:=W;** ***go to if*** **U≥V∧U≥W** ***then*** **ULN** ***else if*** **V≥U∧V≥W** ***then*** **VLN** ***else*** **WLN; ULN:Z:=U; VLN:Z:=V; WLN:Z:=W**

6-34. ***if if*** **V≤U∧V≤W** ***then*** **Z:=V** ***else if*** **U≤V∧U≤W** ***then*** **Z:=U** ***else*** **Z:=W**

6-35. Using a single conditional statement of the first form, write a code that computes $\min(a^3 - bc,\ a - b/c^3)$.

6-36. Using a single conditional statement of the second form, write a code that computes $\min[\max(a,b,c), \max(d,e,f)]$.

6-37. Discuss the difference between the uses of *if* in an expression and in a conditional statement. Give an example that uses both types of *if* in the same statement.

Write codes in the ALGOL language to compute the following five functions:

6-38. $\dfrac{n!}{x!(n-x)!}$.

6-39. $u = \displaystyle\sum_{x=0}^{n} x \frac{n!}{x!(n-x)!} p^x q^{n-x}$.

6-40. $x = \displaystyle\sum_{i=1}^{n} \sum_{j=0}^{i} (i+j)(i-j)$.

6-41. $\arctan x = \dfrac{x}{1+x^2}\left[1 + \dfrac{2}{3}\dfrac{x^2}{1+x^2} + \dfrac{(2)(4)}{(3)(5)}\dfrac{x^4}{(1+x^2)^2} + \cdots\right]$.

6-42. $\arctan x = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{4x^2}{5 + \cfrac{9x^2}{7 + \cfrac{16x^2}{9 + \cdots}}}}}$

6-43. Extend the syntactical definition of a designational expression to include switch lists. Discuss the advantages of such an arrangement.

6-44. Write a syntactical definition of an array declaration.

6-45. Certain parts of the procedure-statement and procedure-declaration syntactical definitions utilize the form ⟨empty⟩. In each case where it appears, give a specific example of how this case may arise.

Write procedures for determining the values of the six following:

6-46. $x = \sqrt{N}$.

6-47. The trace T of a matrix (a_{ij}), where

$$T = \sum_{i=1}^{n} a_{ii}$$

6-48. The transpose of a matrix, where

$$a_{ij}{}^t = a_{ji}$$

6-49. That element of a matrix which has the greatest absolute value.

6-50. The integral A of $f(x)$ to an accuracy of ε by Simpson's rule (see Sec. 9-6).

6-51. An element of a table by the method of Fig. 3-11.

6-52. Write a procedure for ordering a list of numbers (see Secs. 11-5 and 11-6).

Consider the following procedure:

```
real procedure GPS(I,N,Z,V); real I,N,Z,V;
                    begin for I:=1 step 1 until N do Z:=V; GPS:=1 end
```

6-53. Using the **GPS** procedure, write a code for forming the scalar product of two vectors.

6-54. Using the **GPS** procedure, write a code for forming the vector product of two vectors.

6-55. The **GPS** procedure is called (modestly) the "general problem solver" by D. E. Knuth and J. N. Merner, in their "ALGOL 60 Confidential," *Communs. ACM*, vol. 4, no. 6, June, 1961, because, in concept, by using the **GPS** any computable function can be computed. Discuss the use of **GPS** from the point of view of ALGOL as applied to the processes of matrix multiplication and finding the mth prime number as given in that article.

REFERENCES

Arden, B. W., B. A. Galler, and R. M. Graham: An Algorithm for Equivalence Declarations, *Communs. ACM*, vol. 4, no. 7, pp. 310–313, July, 1961.

Backus, J.: Automatic Programming: Properties and Performances of FORTRAN Systems I and II, *Proc. Symposium Mechanization of Thought Processes*, National Physics Laboratory, Teddington, England, H. M. Stationery Office, London, 1959.

Bernick, M. D., E. D. Callender, and J. R. Sanford: ALGK—An Algebraic Manipulation Program, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 389–392.

Bottenbruch, H.: Use of Magnetic Tape for Data Storage in the ORACLE-ALGOL Translator, *Communs. ACM*, vol. 4, no. 1, pp. 15–18, January, 1961.

Cantrell, H. N., J. King, and F. E. King: Logic Structure Tables, *Communs. ACM*, vol. 4, no. 6, pp. 272–275, June, 1961.

Chomsky, N.: Three Models for the Description of Language, *IRE Trans. on Inform. Theory*, vol. IT-2, pp. S113–S124, September, 1956.

———: On Certain Formal Properties of Grammars, *Inform. and Control*, vol. 2, no. 2, June, 1959.

Fortran Programmer's Reference Manual, IBM Corp., New York, Oct. 15, 1956.

Grau, A. A.: Recursive Processes and ALGOL Translation, *Communs. ACM*, vol. 4, no. 1, pp. 10–15, January, 1961.

Green, J., et al.: Remarks on ALGOL and Symbol Manipulation, *Communs. ACM*, vol. 2, no. 9, pp. 25–27, September, 1959.

Harris, Z. S.: "Methods in Structural Linguistics," University of Chicago Press, Chicago, 1951.

Huskey, H. D., and W. H. Wattenburg: Compiling Techniques for Boolean Expressions and Conditional Statements in ALGOL 60, *Communs. ACM*, vol. 4, no. 1, pp. 70–74, January, 1961.

Ingerman, P. Z.: Dynamic Declarations, *Communs. ACM*, vol. 4, pp. 59–60, January, 1961.

Irons, Edgar T.: A Syntax Directed Compiler for ALGOL 60, *Communs. ACM*, vol. 4, no. 1, pp. 51–55, January, 1961.

——— and W. Feurzeig: Comments on the Implementation of Recursive Procedures and Blocks in ALGOL 60, *Communs. ACM*, vol. 4, no. 1, pp. 65–69, June, 1961.

Knuth, D. E., and J. N. Merner: ALGOL 60 Confidential, *Communs. ACM*, vol. 4, no. 6, pp. 268–271, June, 1961.

Lambek, J.: The Mathematics of Sentence Structure, *Am. Math. Monthly*, vol. 65, pp. 154–170, 1949.

Naur, Peter: "A Manual of the DASK ALGOL Language," Regnecentralen, Copenhagen, November, 1960.

——— et al.: Report on the Algorithmic Language ALGOL 60, *Communs. ACM*, vol. 3, pp. 299–314, May, 1960.

Newell, A., F. M. Tonge, E. A. Frigenbaum, G. H. Mealy, N. Saber, B. F. Green, Jr., and A. K. Wolf: Information Processing Language V Manual, secs. I, II, 1960.

Perlis, A. J., and K. Samelson: Preliminary Report—International Algebraic Language, *Communs. ACM*, vol. 1, pp. 8–22, December, 1958.

Sattley, Kirk: Allocation of Storage for Arrays in ALGOL 60, *Communs. ACM*, vol. 4, no. 1, pp. 60–64, January, 1961.

Shaw, J. C.: A Command Structure for Complex Information Processing, *Proc. Western Joint Computer Conf.*, May, 1958.

Smith, J. W.: Syntactic and Semantic Augments to ALGOL, *Communs. ACM*, vol. 3, no. 4, pp. 211–213, April, 1960.

Wilkerson, M.: The JOVIAL Checker, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 397–404.

Woodger, M.: An Introduction to ALGOL 60, *Computer J.*, vol. 3, no. 2, pp. 67–75, July, 1960.

CHAPTER 7

BUSINESS-ORIENTED DATA-PROCESSING LANGUAGES: COBOL

7-1. Common Business Oriented Language COBOL

Introduction. In the previous chapter we studied an algebraically oriented language, ALGOL, which was used to illustrate the concepts involved in the use of the basic operational procedures and subroutines of an integrated, self-consistent automatic-programming language. Attention was particularly focused (1) on the arithmetic, Boolean, and designational expressions as the fundamental ingredients of the procedures; (2) on the basic types of procedures, including the simple "assignment," the "conditional," the repetitive "for," and so forth, as presenting sufficient manipulatory capability to perform any general computation; and (3) on the method of utilizing subroutines ("procedure declarations") to add further variety to the procedures, as the method for developing a rich source of specialized statements for any particular class of problems.

In ALGOL, very little attention is paid to the form or the nature of the data, or to means for successively manipulating large collections of records for systematic processing. The purpose of the present chapter is to study a language that is specifically oriented for such data-handling and -processing tasks. The "*co*mmon *b*usiness *o*riented *l*anguage," called *COBOL*, will be used for this purpose, to illustrate methods (1) for organizing large masses of data and (2) for systematically processing such data in successive iterative manipulations. In addition, the study of COBOL will demonstrate (3) the extensive use of expressions and statements that more closely parallel the clauses, phrases, and statements of ordinary English usage.

COBOL, like ALGOL, is intended to be a generally accepted language, for which compilers are being written that will enable the running of a single COBOL program on many different computers. In May, 1959, representatives from the government, computer manufacturers, and industrial computer users met to form the COBOL committee. By December, 1959, the first COBOL report was completed, and in April,

1960, the U.S. Government Printing Office published "COBOL—A Report to the Conference on Data Systems Languages, Including Initial Specifications for a Common Business Oriented Language (COBOL) for Programming Electronic Digital Computers." This report was revised in 1961, and a report "COBOL-1961" was prepared. The present treatment is based on this "COBOL-1961" revised report. However, the treatment given in this chapter is of a much simplified version of COBOL-1961. The essential concepts involved in the COBOL language can easily be buried in a plethora of nonbasic details and refinements. These refinements become easy to grasp when the fundamental underlying concepts are thoroughly understood. Hence we have chosen to simplify COBOL wherever we felt that pedagogical enhancement would result; we have attempted whenever possible to strip the discussion to its essentials, omitting many refinements and complicating details. The reader should be adequately warned that we are *not* presenting COBOL-1961 in this chapter, but rather we are restricting ourselves to the consideration of its essential concepts; to use COBOL-1961, the reader is referred to the comprehensive treatment of the language given in the "COBOL-1961" report itself or to a computer manufacturer's COBOL manual. However, on consideration of the report or the manuals, it quickly becomes apparent that it is almost always best to approach them with a knowledge of the basic aspects of COBOL, as given in this chapter, as a prerequisite.

Characters and Words. The metalanguage symbolism that we used for ALGOL (see Sec. 6-1) will also be used in the discussion of COBOL. Hence words enclosed within ⟨ ⟩ will be understood as generic terms referencing the COBOL language items; the | is again the metalanguage "or"; and ::= will once more be used for "is defined as being." An additional pair of symbols will also be of some use, namely { }, where the words enclosed are understood to be *alternatives*. For example in COBOL, the operator for add can be denoted by **+** or by **PLUS**; hence we would write

$$\left\{\begin{matrix}\mathbf{+}\\ \mathbf{PLUS}\end{matrix}\right\}$$

As before, direct COBOL words and symbols will appear in boldface.

The *characters* of COBOL are†

⟨letter⟩ ::= **A**|**B**|**C**|**D**|**E**|**F**|**G**|**H**|**I**|**J**|**K**|**L**|**M**|**N**|**O**|**P**|**Q**|**R**|**S**|**T**|**U**|**V**|**W**|**X**|**Y**|**Z**
⟨digit⟩ ::= **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**
⟨logical value⟩ ::= **T**|**F**

† For our discussion of COBOL, we have chosen to reduce the number of characters from that actually used.

⟨arithmetic operator⟩ ::= { + | PLUS } | { − | MINUS } | { × | TIMES } | { / | DIVIDED BY }

⟨relational operator⟩ ::= **IS GREATER THAN**|**IS LESS THAN** | { = | IS EQUAL TO }

⟨logical operator⟩ ::= **AND**|**OR**|**NOT**

⟨delimiter⟩ ::= **(**|**)**| **:**|**.**|**;**|**,**|**"**|**'**

⟨editing symbols⟩ ::= **$**|*|-

A *word* of COBOL is analogous to a *variable* of ALGOL, except that two or more fragments may be combined with a hyphen, -; that is,

⟨fragment⟩ ::= { ⟨letter⟩ | ⟨digit⟩ } | ⟨fragment⟩ { ⟨letter⟩ | ⟨digit⟩ }

⟨word⟩ ::= ⟨fragment⟩|⟨word⟩-⟨fragment⟩

Note that in COBOL a word can start with a digit as well as a letter. The definition of a number for COBOL is exactly the same as that for a number of ALGOL and will not be repeated. However, unlike ALGOL, a word or a number in COBOL is *limited to* 30 *characters*.

In COBOL, several different *kinds* of words are distinguished, in a *semantic* rather than syntactical sense. These are (1) *data-names*, (2) *literals*, (3) *condition-names*, (4) *procedure-names*, and (5) *verbs*. The "data-name" is the name (or address) of a data-item. That is, each of the data-items, which themselves never explicitly appear in a COBOL code but reside on the data tapes, must be denoted by a data-name. In COBOL we say that the *value* of a data-name is a data-item, just as in machine language we say that the *contents* of an address is a word. The COBOL statements generally deal only with the data-names whose *values* are operated upon—in the same manner as machine-language instructions deal only with the addresses whose contents are operated upon. The data-name must be spelled with *at least one* alphabetic character (letter). A data-name may be hyphenated, such as

QUANTITY-IN-STOCK or **STOCK-NUMBER**

There are occasions when it is desirable to include a "data-item" explicitly in the COBOL code itself; such data-items are called *literals*. In other words, a literal is a data-name whose value is identical to the characters comprising the name. For example, in the arithmetic expression **3×Y+6**, the numbers **3** and **6** are literals, but **Y** is the name of some other number in the data. Numbers appearing in a COBOL code will always be interpreted as literals. However, *nonnumerical literals must be placed within quotation marks* so that they can be distinguished from data-

names, e.g.,

"CHAIR" or **"FLIGHT-128"**

It is often desired to give names to the literals themselves, and these are called *condition-names* to distinguish them from data-names. Thus the value of a condition-name is a literal, which appears explicitly in a COBOL code; the value of a data-name is a data-item on the data tapes. As an example of the use of data-names, suppose for a certain problem we desired **A** to stand for **NEW-YORK**, **B** to stand for **TRENTON**, **C** to stand for **WILMINGTON**, and **D** to stand for **BALTIMORE**. Then we would assign

NEW-YORK as the condition-name for the literal **"A"**
TRENTON as the condition-name for the literal **"B"**
WILMINGTON as the condition-name for the literal **"C"**
BALTIMORE as the condition-name for the literal **"D"**

We shall describe below (see Special-purpose Data Descriptions in Sec. 7-4) how condition-names are distinguished from data-names in a very special manner.

Unlike ALGOL, in COBOL the individual statements themselves cannot be labeled. Instead, only a "block" or "paragraph" of statements can be labeled, and such labels are called *procedure-names*. The procedure-name may be composed solely of digits, but most often the procedure-name spells out some mnemonic for the procedure being labeled.

A *verb* is a word, associated with a statement, that usually identifies the type of statement, such as **ADD**, **GO TO**, etc. The verb appears as a fixed part of the syntax of the statement.

Summarizing, the data-name is analogous to the variable of ALGOL, the procedure-name to the label of ALGOL; the concept of the literal, which was implicit in ALGOL for numerical values, is extended to include any word as well, and the names of literals are called condition-names.

The variables or data-names may be *subscripted*, but a maximum of only *three* subscripts are allowed, and the subscripts are limited to only *integers* or data-names and condition-names whose values are integers, as in, for example,

RATE (REGION, STATE, CITY)
RATE (3, 5, 6)

If both these forms represent the same element, then the value of **REGION** must be **3**, the value of **STATE** is **5**, and that of **CITY** is **6**. The use of subscripted variables is the only way of referencing an element of a table, and subscripted variables are interpreted only as refer-

encing elements of a table (see Special-purpose Data Descriptions in Sec. 7-4 for further details).

Over-all Organization. The problems for which COBOL is designed are those that involve the processing of large masses of data. Hence to perform a COBOL computation the computer requires two kinds of information: (1) the *program*, in COBOL language, that is to be executed in order to perform the processing; and (2) the *data* which is being processed (see Fig. 7-1). The actual data is usually repetitive in nature, e.g., records for each of, say, 1,000 checks, payroll-record information for each of 300 employees, sales-slip records for each of 500 sales, etc.

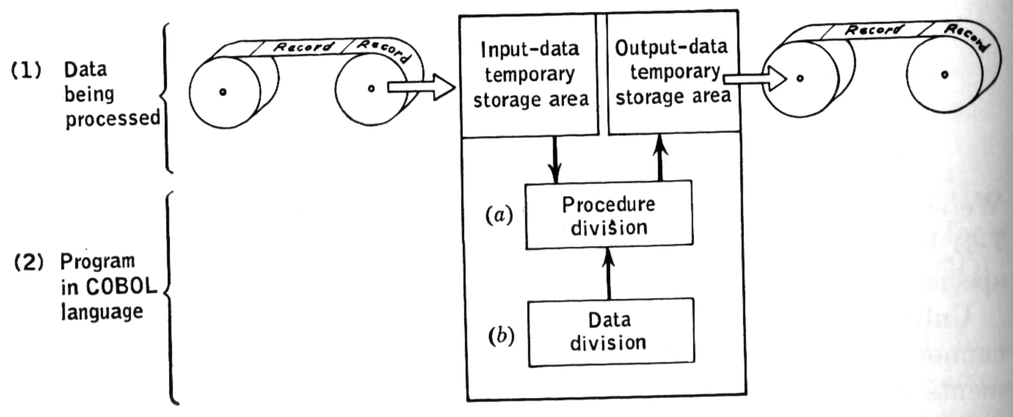


FIG. 7-1. Data and program organization.

Depending on the problem, the successive records of the data are recorded in predetermined formats. For example, a sales-slip format could be as illustrated in Fig. 7-2*a*. The data itself is the information, recorded in the proper format, for example as in Fig. 7-2*b*. Of course, when recorded on magnetic tape, the information is strung out as in Fig. 7-2*c*. In order that the computer should be able to distinguish the different data-items, the computer must be "told" the format, with the different data-items given data-names (see Fig. 7-2*d*). Hence the COBOL-language program must include (*a*) a so-called "data division," where the organization of the data records is specified, as well as (*b*) a "procedure division," where the actual processing operations are specified (see Fig. 7-1). Thus the data division effectively specifies symbolic addresses to the data-items. In this chapter we shall consider both these major divisions of the COBOL language.

Large-scale data processing involves the successive processing of data records, one at a time, in sequence. For example, in the case of the sales slips, for each sale a tally might be kept of how many items of the same type were sold; a total or gross income could be kept; a customer's bill might be made; inventory levels could be adjusted; and so forth.

When the processing has been completed on one sales-slip record, the next record would be read into the computer from the magnetic tape, and so forth. Note that, since the data description given in the data division of the program assumes that the data is in a particular prespecified format, *the data itself must be initially recorded on the magnetic tape in this format.*

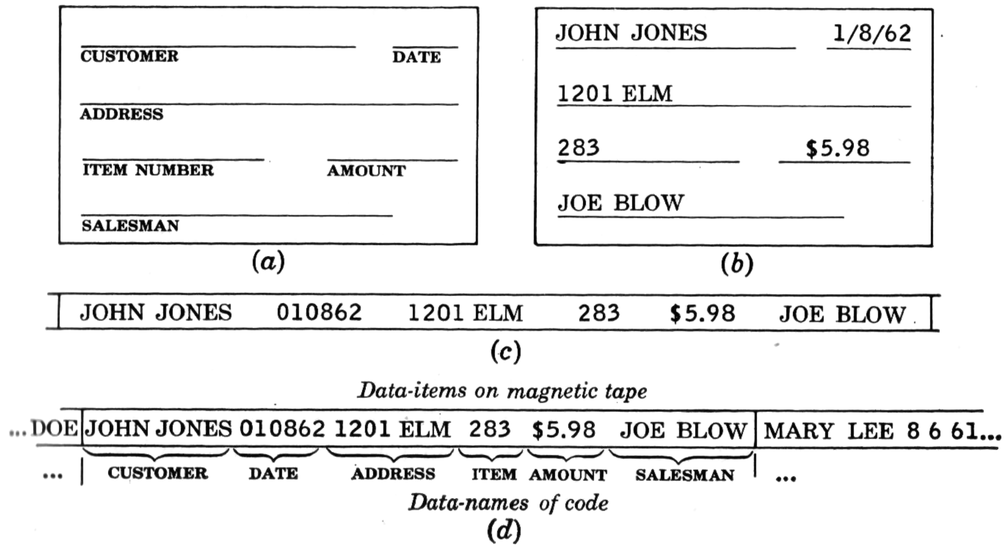


FIG. 7-2. (*a*) Illustration of a sales-slip format; (*b*) data recorded in the sales-slip format of (*a*); (*c*) data strung out for recording on magnetic tape; (*d*) illustration of relation between data-names and data-items.

COBOL differs from ALGOL in two fundamental ways: The first is that COBOL is specially set up to perform the same type of processing repeatedly for successive sets of data. Hence the data-handling aspects of COBOL become of prime importance. The second concerns the fact that the record data itself is distinct from the COBOL-language program or code that is to process it, and is *not* written as, or considered to be, part of the code. *Only data in the form of literals can be part of the code itself.* Separating the data from the code could have the serious disadvantage that the data could not in any way control the processing being done by the code. To avoid such a disadvantage, the concept of the *condition-variable* is introduced. A condition-variable is a data-name whose value is limited to certain specified values, called *option-values*. For example, let **FACTORY** be a condition-variable, and suppose that only four factories, denoted by **A**, **B**, **C**, and **D**, are involved in the problem. Then if each record *on the tape* is associated with a particular factory, each will contain a particular option-value (**A**, **B**, **C**, or **D**) for the condition-variable **FACTORY**. If the code is to perform a different proc-

essing for each factory, then (the "contents" of) **FACTORY** would have to be compared with the literals **"A"**, **"B"**, **"C"**, and **"D"**, which must be written *in the code* itself, and jumps would be made to different sections of the code in accordance with the outcomes of the comparisons. Thus the data on the magnetic tape can influence the processing performed by the code. Often the condition-names of the literals are used instead of the literals themselves. For instance, (the "contents" of) **FACTORY** (i.e., the data-item on the magnetic tape) would be compared with (the "contents" of) **NEW-YORK**—i.e., a literal given in the COBOL-language code—and so forth. In this way a bridge is made between the data on the magnetic tape and a code in COBOL language, by means of the condition-variable and condition-name (or literal) constructs.

Procedure and Data Divisions. A code written in the COBOL language is composed of two parts, the *procedure division* and the *data division.*† The procedure division contains the operational statements, whose execution performs the processing. The data division contains the specifications for and assignments of the data-names, the condition-names, the literals, and other structures to be discussed below. The procedure division involves statements that are similar to those of ALGOL; it will be considered in Sec. 7-2. The data-division description embodies the new concepts contributed by this chapter and is considered in Secs. 7-3, 7-4, and 7-5. In Sec. 7-6 the data-handling methods unique to COBOL (as compared with ALGOL) are considered, and this involves both the procedure and data divisions; at the end of that section a complete COBOL code is illustrated.

In the procedure division, the *statements* in COBOL (which in ALGOL were demarcated by semicolons) are placed together to form *sentences*, analogous to the compound statements of ALGOL. Hence *a sentence is composed of one or more statements separated by semicolons and ends with a period.* A group of successive sentences that may be related in function form a paragraph, which is written with a physical separation. *Only a paragraph can have a label*, called a *procedure-name.* This label is placed as a heading to the paragraph (see page 289 for an example).

The data division is concerned with the *record*, *file*, *environment*, and *identification* descriptions. The *records* are the basic data-items being processed; collections of records are called *files.* By "environment" is meant the physical complement of input-output equipment available for the particular problem, as well as the assignment of files to these equipments. The identification description consists of a simple identification of the code and file tapes.

† COBOL actually requires four divisions: the procedure division, the data division, the environment division, and the identification division. We have chosen to lump these last three together, calling them collectively the *data division.*

7-2. Operational Statements

Boolean and Arithmetic Expressions. The form and nature of the Boolean and arithmetic expressions of COBOL are analogous to the *simple* forms of the corresponding expressions of ALGOL, but words are sometimes used for relational and arithmetic operations. The relations of COBOL, however, are not limited to comparisons of magnitudes of numbers but include comparisons of *words as literals* as well. Here "**IS GREATER THAN**," "**IS LESS THAN**," and "**IS EQUAL TO**" for words are understood as if the words involved were numbers with the radix 26. For example, **"ABAA" IS GREATER THAN "AAAD"** is true; and **"DOG" IS EQUAL TO "DOG"** is true [where we have put the distinguishing quotation marks (", ") around the literals]. For words of unequal length, the right-hand part of the shortest word is filled with "blanks" to make the words of equal length, with the blank considered smaller than any letter. Thus **"ABAA" IS GREATER THAN "ABA"** is true. In general, condition-variables and condition-names (that is, words whose values are respectively data-items and literals) are often used in the relations, rather than the literals themselves. For instance, if it were desired to determine whether or not a report were from the New York factory, a comparison of the type (the "contents of") **FACTORY IS EQUAL TO** (the "contents of") **NEW YORK** would be made, where **FACTORY** is a conditional-variable and **NEW YORK** is a condition-name. For relations, then, the operations involved are

⟨relational operation⟩ ::= **IS GREATER THAN**|**IS LESS THAN**|{ **=** | **IS EQUAL TO** }

Hence a relation can be given the following syntactical definition:

⟨relational operand⟩ ::= ⟨literal⟩|⟨data-name⟩|⟨arithmetic expression⟩

⟨relation⟩ ::= ⟨relational operand⟩ ⟨relational operation⟩ ⟨relational operand⟩

Of course both operands cannot be literals, or the test would be trivial; rather, if one is a literal, the other must be a data-name of a literal which is looked up (evaluated) for the comparison.

The Boolean expression, as in ALGOL, is a rule for computing a logical value, *true* or *false.* Boolean expressions of relations involve logical operations, namely,

⟨logical operation⟩ ::= **OR**|**AND**|**NOT**

and can be formed like the simple Boolean expression of ALGOL—with

the same precedence order of operations, namely,

First.	Arithmetic expressions
Second.	Relational operations
Third.	**NOT**
Fourth.	**AND**
Fifth.	**OR**

The COBOL language allows certain displacements and omissions of words, in order that it correspond more closely to customary English usage. The negation, or **NOT**, is usually placed after the **IS** in a relation; e.g.,

NOT (A IS LESS THAN B)

becomes

A IS NOT LESS THAN B

The left operand can be omitted from a relation in instances where it would normally be understood, such as where

A IS LESS THAN B OR A IS EQUAL TO C

becomes

A IS LESS THAN B OR EQUAL TO C

Such omissions can be compounded, as in

A IS EQUAL TO B AND C OR IS LESS THAN D

The general rule used in COBOL *in all cases* is that the compiler takes the missing parts from the *nearest preceding full relation which explicitly states all three parts in the same sentence*, no matter what semicolons, parentheses, connectives or other words, etc., may lie in between. This rule implies that the first relation of a sentence must have all the parts explicitly stated (why?).

Arithmetic expressions can likewise be expressed in terms of words for the operations, although here the symbols may be used if desired. The order of precedence of the operations is the same for COBOL as for ALGOL:

Order	Symbol†	Word equivalent
First.....	↑	**EXPONENTIATED BY**
Second.....	×, /	**TIMES, DIVIDED BY**
Third.....	+, −	**PLUS, MINUS**

† COBOL actually uses ** instead of ↑ and * instead of ×. The relation **IS EQUAL TO** can as well be symbolized by =, as we have seen.

The *arithmetic expression* is, as in ALGOL, a rule for computing a numerical value; its syntactical definition for COBOL is identical to that of the simple arithmetic expression of ALGOL.

Assignment Class of Statements. Corresponding to the assignment statement of ALGOL, there is a *class* of statements of COBOL. The first is called the **COMPUTE** statement and has the following syntax:

⟨left part compute⟩ ::= **COMPUTE** ⟨data-name⟩ { **FROM** | **=** | **EQUALS** }

⟨compute statement⟩ ::= ⟨left part compute⟩ ⟨arithmetic expression⟩

Here the words **FROM** and **EQUALS** are equivalent to each other and to =. The effect of the statement is to evaluate the arithmetic expression and assign its value to the data-name of the left part. (When the arithmetic expression is itself just a data-name, the effect is to assign its value to the data-name of the left part.) For example,

COMPUTE PRICE FROM COST TIMES (ITEMS MINUS DISCNT)

would multiply the value of **COST** by the value of **ITEMS — DISCNT** and assign the resulting value to **PRICE**.

The term **COMPUTE** is called a *verb* of COBOL, for it describes an action to be performed. Another verb in the assignment-statement class is **ADD**; the **ADD** statement has the following syntax:

⟨addend⟩ ::= ⟨literal⟩|⟨data-name⟩

(where of course the "literal" must be a number)

⟨addend list⟩ ::= ⟨addend⟩ | ⟨addend list⟩, ⟨addend⟩

⟨left part add⟩ ::= **ADD** ⟨addend⟩, ⟨addend list⟩

⟨add statement⟩ ::= ⟨left part add⟩|⟨left part add⟩ { **TO** | **GIVING** } ⟨data-name⟩

Thus the **ADD** statement has three forms: (1) the first has just a list of two or more addends (why?); (2) the second has a "left part add" with the word **TO** followed by a data-name; and (3) the third has a "left part add" with the word **GIVING** followed by a data-name. The computations to be performed for each case are as follows: (1) The sum of the values of all the addends is formed and assigned as the value of the rightmost addend (which therefore must be a data-name). (2) The sum of the left-part addend values is added *to* the data-name following **TO** and becomes the new value of that data-name. (3) The sum of the

left-part addend values becomes the value of the data-name following the **GIVING**. For example, to add the **PRICE** and the **SHIPPING** costs and a sales tax of $3.85 forming the **TOTAL**, we would have the following for each of the forms:

```
ADD PRICE, SHIPPING, 3.85, TOTAL.
ADD PRICE, SHIPPING, 3.85 TO TOTAL.
ADD PRICE, SHIPPING, 3.85 GIVING TOTAL.
```

In the first two cases, the value of the **TOTAL** would be set to zero initially; this is not necessary for the third case.

The rest of the statements of the assignment class are similarly constructed. Thus **SUBTRACT** is syntactically defined as

⟨left part sub⟩ : : = **SUBTRACT** ⟨addend⟩|**SUBTRACT** ⟨addend list⟩

⟨subtract statement⟩ : : = ⟨left part sub⟩ **FROM** ⟨data-name⟩
|⟨left part sub⟩ **FROM** ⟨addend⟩ **GIVING** ⟨data-name⟩

The value of the addend, or of the *sum* of addends, of the "left part sub" is subtracted from the value of the data-name or addend following the **FROM**, and this difference becomes the new value of the data-name. For example, to form an **INVOICE** as the **PRICE** less the **DISCOUNT** and a credit of $3.85, we have

```
SUBTRACT DISCOUNT, 3.85 FROM PRICE GIVING INVOICE.
```

For **MULTIPLY**, we have

⟨multiply statement⟩ : : = **MULTIPLY** ⟨addend⟩ **BY** ⟨data-name⟩
|**MULTIPLY** ⟨addend⟩ **BY** ⟨addend⟩ **GIVING** ⟨data-name⟩

Similarly for **DIVIDE** we have

⟨divide statement⟩ : : = **DIVIDE** ⟨addend⟩ **INTO** ⟨data-name⟩
|**DIVIDE** ⟨addend⟩ **INTO** ⟨addend⟩ **GIVING** ⟨data-name⟩

For example, suppose we wished to find the **PRICE** of 36 **ITEMS** listed at $150 per dozen, where we get a 10 per cent **DISCOUNT**:

```
DIVIDE 12 INTO ITEMS GIVING DOZENS;
MULTIPLY DOZENS BY 150 GIVING COST;
MULTIPLY COST BY DISCOUNT GIVING AMOUNT;
SUBTRACT AMOUNT FROM COST GIVING PRICE.
```

GO TO *and* **ALTER** *Statements.* These COBOL statements are analogous to the ALGOL ***go to*** statement and ***switch*** declaration. Consider

first the COBOL **GO TO** statement, which has two forms:

⟨label list⟩ : : = ⟨label⟩|⟨label list⟩, ⟨label⟩

⟨go to statement⟩ : : = **GO TO** ⟨label⟩
|**GO TO** ⟨label⟩, ⟨label list⟩ **DEPENDING ON** ⟨data-name⟩

The first form is clear. In the second form, there is a choice of two or more destinations. The value of the data-name must be a positive integer between 1 and $N + 1$, where N is the number of labels in the label list; then the first label corresponds to 1, and the labels of the label list in order from left to right correspond to the numbers 2 through $N + 1$. The effect of the **GO TO** statement will be to jump to that paragraph which corresponds in order number to the value of the data-name.

The **ALTER** statement is as follows:

⟨alter statement⟩ : : = **ALTER** ⟨label⟩ **TO PROCEED TO** ⟨label⟩

The **ALTER** statement acts only on a **GO TO** statement *with a single destination label.* Its effect is to change the destination. The **GO TO** statement to be changed must form a paragraph by itself, so that it may have its own identification label. Then the label after the word **ALTER** is the identification label of such a **GO TO** statement; the destination label of the **GO TO** statement is replaced by the label that follows the words **TO PROCEED TO** in the **ALTER** statement.

Conditional Statements. The conditional statement of COBOL is of a slightly different form from that of ALGOL. The syntax is

⟨if statement⟩ : : = **IF** ⟨Boolean expression⟩ **THEN** ⟨statement⟩
⟨if fragment⟩ : : = ⟨if statement⟩|⟨if fragment⟩**;** ⟨statement⟩
⟨else statement⟩ : : = **ELSE** ⟨statement⟩
⟨else fragment⟩ : : = ⟨else statement⟩|⟨else fragment⟩**;** ⟨statement⟩
⟨conditional statement⟩ : : = ⟨if fragment⟩ ⟨else fragment⟩ { ⟨empty⟩ | . }

The "empty" alternative is used only when another "else fragment" follows (i.e., when the conditional statement is the final "statement" of an "if fragment"—see example below); otherwise the period *must* be used to indicate the final statement of the "else fragment" (why?). Note that unlike ALGOL the **ELSE** *must occur;* that unlike ALGOL the "if statement" may include a conditional statement; that unlike ALGOL the Boolean expression and other expressions occurring in the statements *cannot* include an **IF** (why?); and that unlike ALGOL the statements within the conditional statements *cannot* be labeled (why?). Because of these first two differences, the interpretation of conditional statements

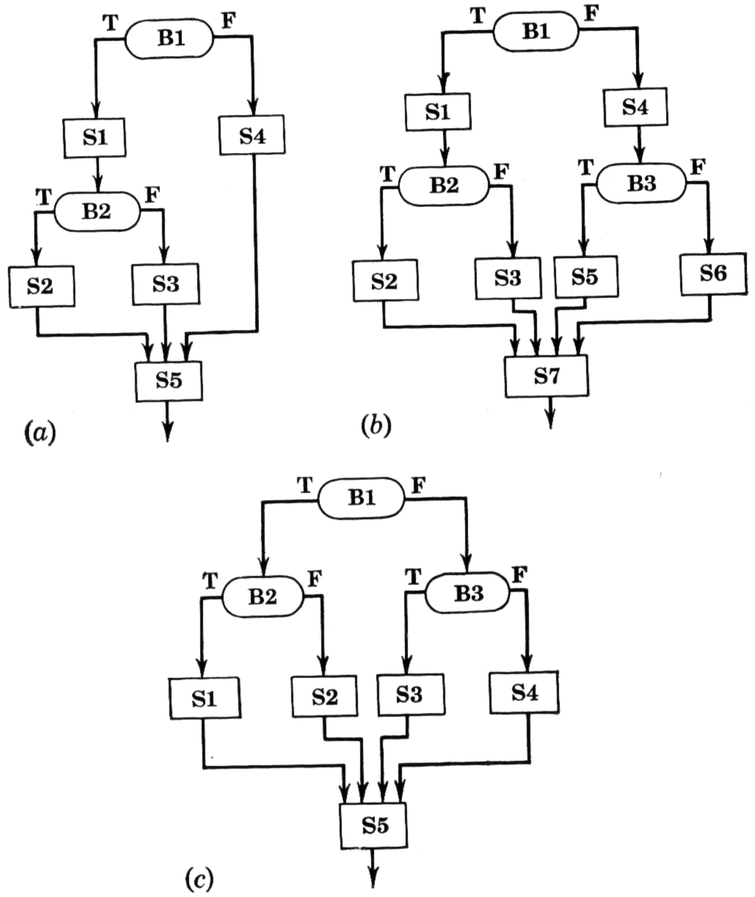


FIG. 7-3. Flow-chart interpretations of conditional statements (see text).

which involve conditional statements is quite different from that in ALGOL. For example, the following conditional statement is interpreted as in Fig. 7-3*a* (cf. Fig. 6-4):

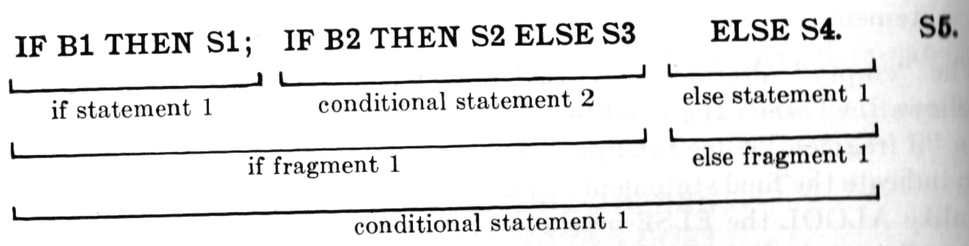
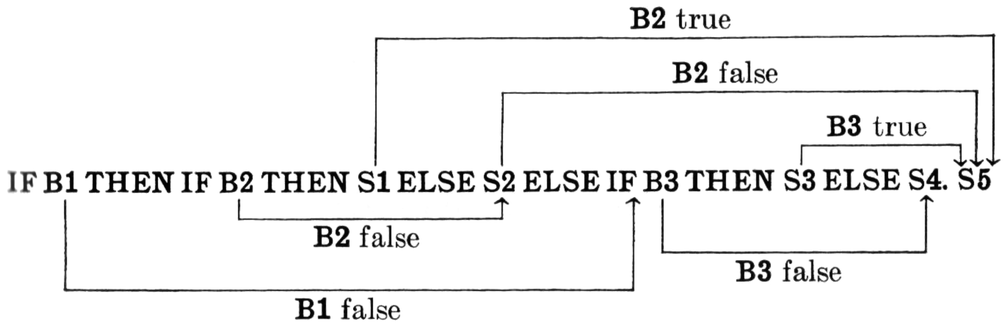


Figure 7-3*b* is the interpretation of the following:

IF B1 THEN S1; IF B2 THEN S2 ELSE S3 ELSE S4;
IF B3 THEN S5 ELSE S6. S7

Figure 7-3*c* is the interpretation of the following:



That these results read something like English can be seen from this example:

IF CUSTOMER-CREDIT IS EQUAL TO GOODRISK THEN
IF CUSTOMER-YRS IS GREATER THAN 5 THEN
COMPUTE DISCOUNT = OLDDISC
ELSE COMPUTE DISCOUNT = NEWDISC
ELSE GO TO BADCUSTOMER.

Here **CUSTOMER-CREDIT** can have the value **GOODRISK** or **BADRISK**; **CUSTOMER-YRS** has the value of the number of years the person named has been a customer; the compute statements fix the value of the discount; **BADCUSTOMER** is the label of a paragraph that describes how to deal with bad customers.

for* Statement.** The loop repetition, or ***for statement, of COBOL is actually denoted as **PERFORM**. The syntax is

⟨left part perf⟩ ::= **PERFORM** ⟨label⟩|**PERFORM** ⟨label⟩ **THRU** ⟨label⟩
⟨from list⟩ ::= **FROM** ⟨addend⟩ **BY** ⟨addend⟩ **UNTIL** ⟨relation⟩
⟨simple tally / condition⟩ ::= ⟨empty⟩|⟨addend⟩ **TIMES**|**UNTIL** ⟨relation⟩
|**VARYING** ⟨data-name⟩ ⟨from list⟩
⟨subscript tally condition⟩ ::= **VARYING** ⟨subscript-name⟩ ⟨from list⟩
| ⟨subscript tally condition⟩ **AFTER** ⟨subscript-name⟩ ⟨from list⟩
⟨perform statement⟩ ::= ⟨left part perf⟩ ⟨simple tally condition⟩
|⟨left part perf⟩ ⟨subscript tally condition⟩

These syntactical rules give five options to the **PERFORM** statement, the first four associated with the four "simple tally condition" alternatives, the fifth concerned with the "subscript tally condition."

Consider first the "empty" option of the simple tally condition: the **PERFORM** statement could then be of the form

PERFORM LABELX THRU LABELY.

which would simply tell the computer to jump to **LABELX**, to execute the statements of that paragraph and the next successive paragraphs until the statements of paragraph **LABELY** were executed, and then to jump back to the original program—no loop or iteration is called for here. Second, if the **PERFORM** statement is of the form

PERFORM LABELX THRU LABELY N TIMES.

then the sequence of paragraphs from **LABELX** through **LABELY** would be repeatedly executed exactly **N** times—here we have a simple loop. Third, the form

PERFORM LABELX THRU LABELY UNTIL A IS LESS THAN B.

tells the computer to execute the sequences of paragraphs repeatedly until the relation **A IS LESS THAN B** (in this case) is true. This is directly analogous to the ***while*** alternative of the ALGOL ***for*** statement. Fourth, we have the analogy to the ***step-until*** option of ALGOL, which in COBOL is really a combined ***step-until*** and ***while*** statement, of the form

PERFORM LABELX THRU LABELY VARYING DANA FROM FIRSTN BY DELTA UNTIL A IS LESS THAN B.

Here the variable **DANA** is explicitly displayed, as are the initial condition **FIRSTN**, the increment **DELTA** to be added to the variable after each iteration, and the exit condition (in this case, "**A IS LESS THAN B**"). The fifth option involves the use of subscripted variables for the formation of loops within loops. Since in COBOL subscripted variables are limited to *only three* subscripts, a maximum of three "from lists" can be involved in the "subscript tally condition." As an example, Fig. 7-4 shows the loops generated by the following statement:

PERFORM LABELX THRU LABELY
VARYING SUBS1 FROM FIRSTS1 BY DELTA1 UNTIL ⟨relation 1⟩
AFTER SUBS2 FROM FIRSTS2 BY DELTA2 UNTIL ⟨relation 2⟩
AFTER SUBS3 FROM FIRSTS3 BY DELTA3 UNTIL ⟨relation 3⟩.

Procedure Statements and Declarations. Two statements of COBOL are used to incorporate subroutines into a program: the **INCLUDE** statement and the **DEFINE** statement. The **INCLUDE**, to which the ALGOL procedure statement is the most closely related, has this syntactical form:

⟨replacement⟩ : : = ⟨data-name⟩ **BY** ⟨data-name⟩
⟨replacement list⟩ : : = ⟨replacement⟩|⟨replacement list⟩, ⟨replacement⟩
⟨include statement⟩ : : = **INCLUDE** ⟨label⟩, **REPLACING** ⟨replacement list⟩

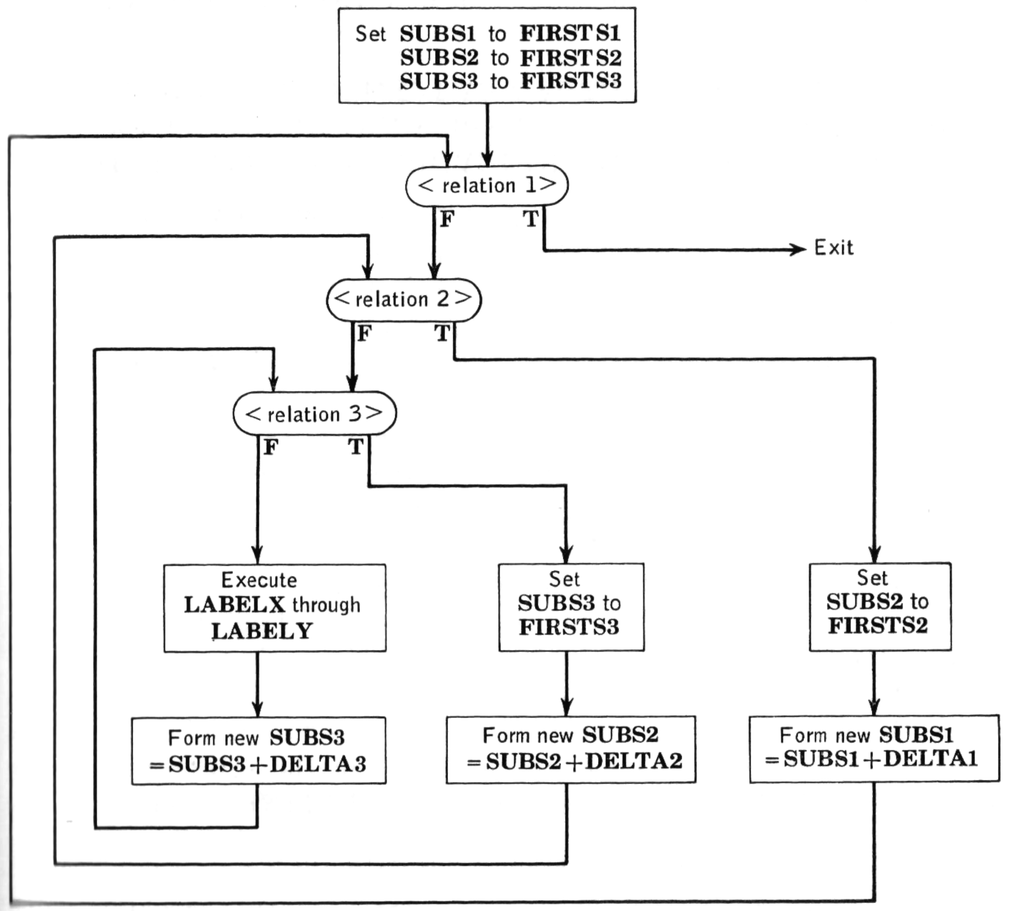


FIG. 7-4. Use of subscripts to form loops by means of the **PERFORM** statement.

The concept of a subroutine in COBOL is simply that of a *labeled paragraph.* The **INCLUDE** statement calls for the subroutine by its label; then before the subroutine can be executed by the program, the variables or data-names of the subroutine corresponding to "independent" or "dependent" variables must be replaced by the corresponding data-names of the program. This correspondence is given in the "replacement list," in which the left-hand data-name of a "replacement" is the name of a variable in the subroutine (as originally written) that is to be replaced by the right-hand data-name of the program. This is analogous

to the role of *formal* and *actual* variables in ALGOL. As an example, consider the following subroutine, labeled **TAKE-DISCOUNT**:

TAKE-DISCOUNT. MULTIPLY DISCOUNT-RATE BY COST GIVING DISCOUNT; SUBTRACT DISCOUNT FROM COST GIVING NET.

The following program is an example of how **TAKE-DISCOUNT** might be used:

MULTIPLY NUMBER-ORDERED BY ITEMCOST GIVING GROSS; INCLUDE TAKE-DISCOUNT, REPLACING DISCOUNT RATE BY 0.05, COST BY GROSS, NET BY INVOICE; ADD INVOICE, TOTAL.

Here the program used the terms **GROSS** and **INVOICE**, which corresponded to **COST** and **NET** of the subroutine. The most important aspect of the replacements is not just the substitution of terms, but is rather the fact that such a substitution of terms means that the *values utilized* in the subroutine execution, as called for by an **INCLUDE** statement, will be those of the program, as desired.

The **DEFINE** statement presents the opportunity for defining *new* statements *in a prescribed format.* The **INCLUDE** statement affords a means for using a subroutine, but the **INCLUDE** statement format must be utilized. With the **DEFINE** statement, the format of a *new statement* may be defined, and then this new statement itself would be used in a program to jump to the subroutine. Hence the English connotation of the subroutine use can be included in its "calling-up" statement. The **DEFINE** statement, which defines this format, is therefore really a true *declaration* rather than a statement; it declares the format of a new statement whose interpretation will be by means of the subroutine of the declaration. The syntactical definition of the **DEFINE** statement is

⟨format⟩ : : = **FORMAT** ⟨verb-name⟩ ⟨verb structure⟩

⟨define statement⟩ : : = **DEFINE** ⟨verb-name⟩ **WITH** ⟨format⟩ ⟨verb subroutine⟩

where by "verb-name" we mean the identification given to the new verb, analogous to the verbs **PERFORM**, **ADD**, **SUBTRACT**, etc.; by "verb structure" we mean the syntactical definition of the new verb; and by "verb subroutine" we mean the subroutine that is called for when the new verb is used in a program. The independent and dependent variables of the verb subroutine must appear somewhere in the "verb structure." The verb structure is written out in full, using some generic identifications for these variables, and the verb subroutine will use these generic identifications also. When the new verb is used in a program, the generic identifications will be replaced by their actual values; when the jump to the new verb subroutine is made, the generic identifications are replaced by the corresponding actual values from the program.

7-3. Over-all Data Organization

Orientation. The operational statements of COBOL were in general analogous to those of ALGOL; however, it is in the organization and handling of data that the primary purpose of COBOL becomes apparent. The data-organization constructs are of central importance in the COBOL concepts. In the over-all view of this data organization, it is assumed that magnetic-tape units are used for storing the files and associated records being processed (see Fig. 7-5). In general, the data itself consists of lists of records, where there is one record of information for each employee, sale, invoice, inventory item, production item, individual medical record, etc., under consideration. Each type of record used is given a fixed *record format,* such as that illustrated in Fig. 7-6. The *record itself* consists just of the information (as, for example, in Fig. 7-7), which becomes meaningful only when the record format is known. A *file* is a collection of similar records, e.g., the collection of records for all the employees of a certain plant, or for all the sales in a fiscal year, or for all the monthly invoices of a business, or for all inventory items of a particular warehouse, or for all items produced by a plant in a week, or for all the medical records of a hospital for a year, etc. Each file will contain only records of a single type (i.e., format), and each file is stored on its own separate magnetic tape.†

Some of the files are *input files:* that is, they supply the initial data to be updated or processed by the computer. The processed information is dumped onto *output files* (see Fig. 7-5). For example, in a payroll processing job, the input files will contain the records of the employees, with the record for each employee containing all the information needed to make up his paycheck. The output files would be the records of what information should be printed on each check and would also be new employee records if some items such as "total earnings to date" must be updated. As a general rule, in updating a file a new file tape is made, rather than the old file tape changed. This is done in case of an error, so that the old and new records may both be available. At some later time, the old file can be erased and the magnetic tape reused. Another example of this is the updating of inventory records, for which

† Actually, each file may contain several different types of records; and it is possible to have more than one file on the same magnetic tape, although it is more common to have more than one tape associated with the same file. These generalizations are easily made at a later stage in the discussion, for they add no new concepts to our exposition. Hence for the present we shall consider the simplest case of a single type of record in a file and one magnetic-tape unit for each file under consideration.

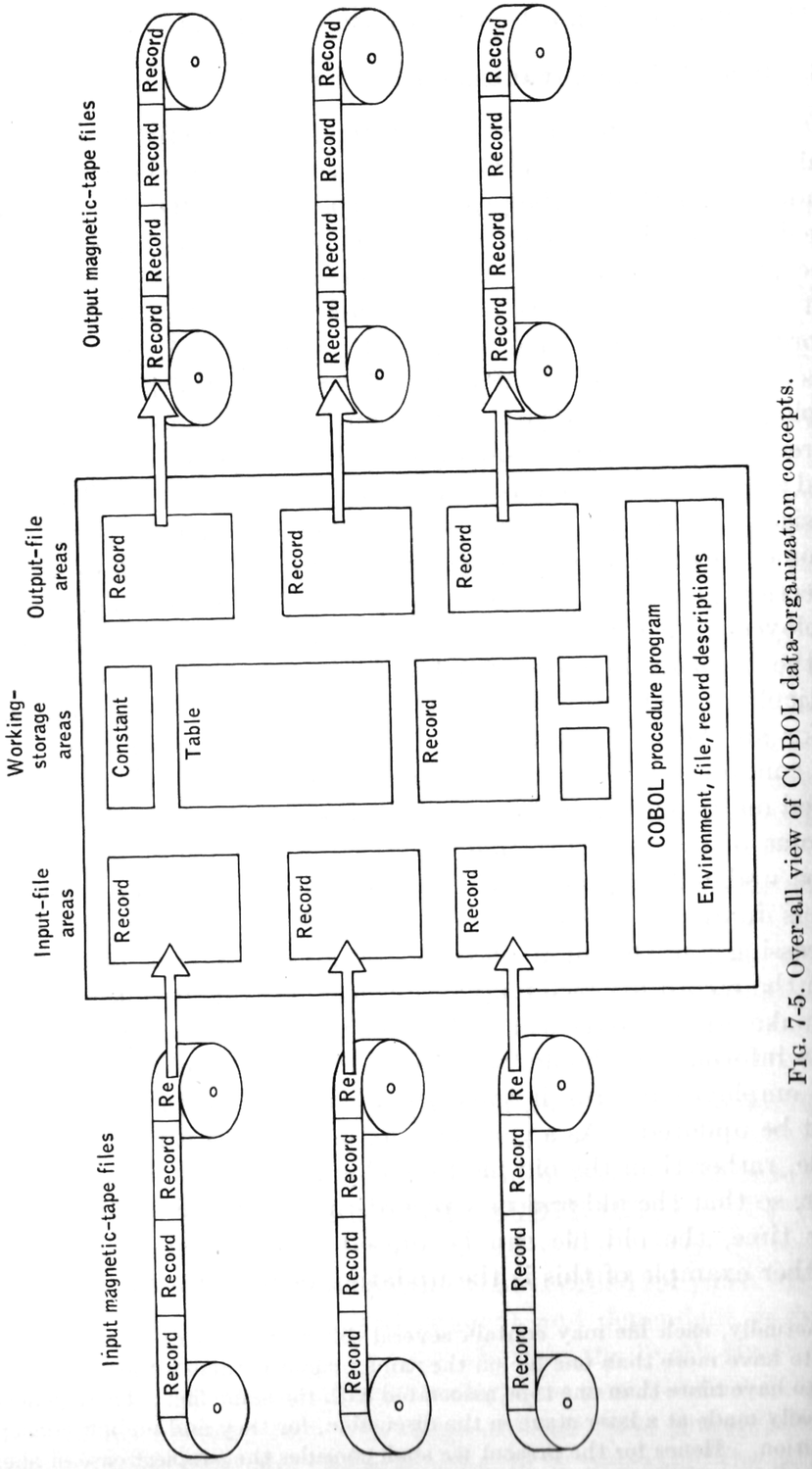


FIG. 7-5. Over-all view of COBOL data-organization concepts.

a new, revised inventory-record file is made, on a new output tape, from the old inventory tape and stock-record changes as input files.

Flow of Information during Processing. The dynamics of a large-scale data-processing problem of this type are as follows: First, note that in general the same kind of processing is to be performed repeatedly on each record of the same type. For example in the payroll case, for each employee, say, on a regular salary, the deductions for social security, Federal income-tax withholding, state income-tax withholding, health insurance, etc., must be made, according to the employee's total earnings

NAME (LAST, FIRST, MIDDLE INITIAL)	MALE OR FEMALE
DATE OF BIRTH (DAY, MONTH, YEAR)	CITIZENSHIP (COUNTRY)
ADDRESS (NUMBER, STREET, TOWN OR CITY, STATE)	
OCCUPATION / SOCIAL SECURITY NUMBER / MARRIED OR SINGLE	

FIG. 7-6. Illustration of the concept of a record description or format.

JONES MARY E.		F
28 JUNE 1940		USA
10 DOWNING ST. HYATTSVILLE MD.		
WELDER	875-65-1935	M

FIG. 7-7. Illustration of record information.

to date, number of exemptions, type of insurance policy, etc. Thus the same computer code will successively process each employee's record; the record must be updated with total earnings to date, with the paycheck data and deductions listed on separate output files. Hence the individual computer records are processed one at a time. For such successive computations, an employee's record is read into the high-speed memory of the computer from the input-file tape; the processing is performed on this record and the several output records are generated; next these are read out of the high-speed memory of the computer onto the proper output-file tapes; then the computer is ready to read in the next employee's record and repeat the processing (see Fig. 7-8).† The computer or the

† Of course it may be more efficient to read (write) more than one record at a time from (onto) the magnetic-tape files, and this is actually done. However, the processing clearly will in general involve one record at a time from a file, and hence without loss of generality we may talk *as if* one record at a time is utilized from a file: indeed, the COBOL instructions are set up to maintain this point of view (see the **READ** and **WRITE** statements in Sec. 7-6).

COBOL code must therefore contain (1) a part that performs the processing on a record and (2) a part that reads in the next record. The instructions for the former part have already been discussed in the previous section; the instructions for the latter data-handling operation will be described in a later section. The purpose of this and the next section is to describe how the computer is told about the format of the records (*the record description*) and about the files or tapes at its disposal (*the file and environment description*)—i.e., to describe the data-organization.

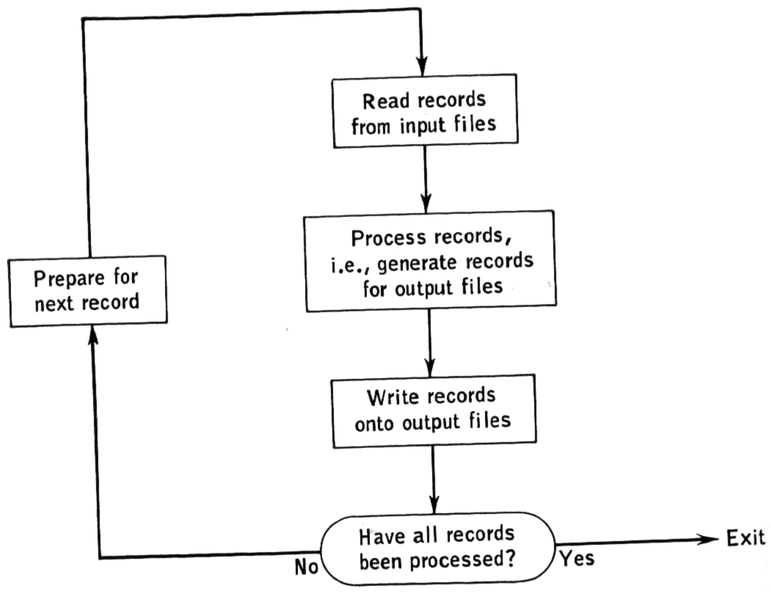


FIG. 7-8. Over-all view of the dynamics of successive record processing.

Consider next the high-speed memory of the computer. As each of the records is read in from an input file, one record at a time, it must be located somewhere in the high-speed memory. Similarly, after an output record is formed, but before it is written onto its file, it must be stored somewhere in the high-speed memory. In addition, during the processing certain other areas of the high-speed memory must be allocated for tables, constants, intermediate record results, etc. These groups of areas (see Fig. 7-5) are called, respectively, the *input-file* areas, the *output-file* areas, and the *working-storage* areas. The computer must be told where and how much space to allot to these areas (by the *record* and *working-storage descriptions*).

The Record Description. The description of a type of record format is accomplished in the *data division* of the COBOL language. Of course, only one record description for each different type of record is needed. A record description tells the computer such things about the organization and nature of the record as the categories and subcategories, the data-names for these categories and subcategories, the size of the actual

data field (i.e., the number of bits or characters a data-item will require), the class or type (e.g., numerical, alphabetical, alphanumerical) of each item, and so forth. The computer requires this knowledge in order (1) to allot the necessary space in the input- and output-file areas for the files, (2) to locate data by its data-name, and (3) to perform the proper type of operations (e.g., arithmetical, logical, etc.). Note from Fig. 7-5 that the record descriptions are always kept in the COBOL area of the high-speed memory, whereas the data-records themselves are on the file tapes, being read into the high-speed memory from each tape, one at a time, only when their processing is taking place.

The format for writing a record description (which is itself a format) is as follows:†

⟨ data-item description ⟩ ::= ⟨level number⟩ ⟨data-name⟩; ⟨data description⟩.

⟨record description⟩ ::= ⟨data-item description⟩|
⟨record description⟩
⟨data-item description⟩

The remainder of this section and the next section are devoted to discussions of the terms mentioned in this syntactical definition. However, it might first be of interest to observe the following example of a record description:

```
1 MASTER; SIZE IS 22 CHARACTERS; CLASS IS ALPHANUMERIC.
    2 BDG-NBR; SIZE IS 7 CHARACTERS; PICTURE IS AAXX999.
    2 DATE; SIZE IS 6 CHARACTERS; CLASS IS NUMERIC.
        3 MONTH; SIZE IS 2 CHARACTERS.
        3 DAY; SIZE IS 2 CHARACTERS.
        3 YEAR; SIZE IS 2 CHARACTERS.
    2 PLT-NBR; SIZE IS 3 CHARACTERS; CLASS IS NUMERIC.
    2 GROSS-PAY; SIZE IS 6 CHARACTERS; PICTURE IS 999.99.
```

Each row of this example represents a "data-item description"; the leftmost digit is the "level number," the leftmost word is the "data-name," and the rest is the "data description."

The level number is used to arrange the data in a hierarchical structure: the higher the number, the deeper the structure. Thus level number **1**

† In this brief and introductory treatment we must necessarily omit some of the record-description entities; however, we hope to treat here the basic concepts involved, with further details to be obtained from the COBOL manual itself. In our syntactical notation, aligned *vertical* juxtaposition of generic entities will imply a columnar listing. Thus

⟨record description⟩ ::= ⟨record description⟩
⟨data-item description⟩

means that a record description is a listing of a column of data-item descriptions.

gives the *name of the record* itself, such as **MASTER**, **TIME-CARD**, **BILL-OF-SALE**, **PROCUREMENT-ORDER**, and so forth. The number-**2** levels give categories of data within the record; the number-**3** levels give categories of data within the number-**2** level under which they are listed; the number-**4** levels give categories of data within the number-**3** level under which they are listed; and so forth. Thus the record **MASTER** (see above) has four subcategories, namely **BDG-NBR**, **DATE**, **PLT-NBR**, and **GROSS-PAY**. The **DATE** itself has three subcategories, **MONTH**, **DAY**, and **YEAR**. As an example of an actual record as it would appear on magnetic tape in the format described above, we could have

Elementary data-item	*Data-names*	
BQY7123	**BDG-NBR**	**MASTER**
06	**MONTH**	
28	**DAY**	**DATE**
61	**YEAR**	
284	**PLT-NBR**	
236.65	**GROSS-PAY**	

where the left-hand column is the actual record data and the boldface words are the respective data-names. Of course, on the magnetic tape the data of a record just appears as a string of letters and numbers, i.e.,

⟨previous record data⟩BQY7123062861284236.65⟨next record data⟩

and the computer, by means of the COBOL compiler, would count the characters to locate the individual items of data. In general, these individual items of data are called the *elementary data-items*.

Very often the data-names of elementary items cannot be used directly but must be qualified by the data-names of lower-numbered (i.e., higher) levels. Such cases arise when an elementary data-item has the same data-name as another elementary data-item in either the same record or another record that will be used in the high-speed memory at the same time. For example, consider the following record descriptions:

```
1 MASTER . . . .
    2 CURRENT-DATE . . . .
        3 MONTH . . . .
        3 DAY . . . .
        3 YEAR . . . .
    2 LAST-TRANSACTION-
      DATE . . . .
        3 MONTH . . . .
        3 DAY . . . .
        3 YEAR . . . .
```

```
1 NEW-MASTER . . . .
    2 CURRENT-DATE . . . .
      →3 MONTH . . . .
        3 DAY . . . .
        3 YEAR . . . .
    2 LAST-TRANSACTION-
      DATE . . . .
        3 MONTH . . . .
        3 DAY . . . .
        3 YEAR . . . .
```


To avoid ambiguity the **MONTH** contained in the **CURRENT-DATE** of the **NEW-MASTER** record (see arrow above) must be referred to as

MONTH IN CURRENT-DATE IN NEW-MASTER

This entire phrase would be used for the data-name of the elementary data-item with the arrow above, where each **IN** is followed by a lower-level-number data-name, successively as illustrated, until no ambiguity can exist. In any case where no ambiguity in data-name can exist, the data-name of the elementary data-item is used directly.

7-4. Detailed Data Descriptions†

Description of Physical and Operational Characteristics. We come now to a discussion of the "data description" part of the record description (see the syntactical definition above). So that the computer will know what to expect as it goes through the actual record data, as well as how to handle the different types of data, certain characteristics of the data can be described, with the descriptions separated by semicolons. The following can be specified concerning the physical and operational characteristics of an item of data:†

SIZE PICTURE CLASS SIGNED USAGE

The **SIZE** clause is specified as

SIZE IS ⟨integer⟩ **CHARACTERS**

where "integer" gives the actual number of characters of the data being described. If the data being described is distributed over deeper (higher-numbered) sublevels, then the total number of characters in these levels is given. For example, if **DATE** is composed of **MONTH**, **DAY**, and **YEAR**, each of which has two characters, then for **DATE** the **SIZE IS 6 CHARACTERS**. This concept of data size is generalized to any number of sublevel categories.

The **PICTURE** clause can be given only for an elementary data-item and is simply a detailed format of its composition in terms of types of characters. In the **PICTURE**, **A** represents an alphabetical character, **X** an alphabetical or numerical (alphanumerical) character, **9** a numerical character, **S** the + or − sign, **.** an actual decimal point, **B** a blank space, and **$** an actual dollar sign.† Then, for example,

PICTURE IS AAAXXX999

† Since our primary aim is to treat *concepts* only, we shall omit from discussion many details and descriptive parts that are not thought pedagogically important for the student at this point. Once the concepts are known, these details and omitted parts may be easily obtained from the COBOL manual itself.

represents data consisting of (from left to right) three alphabetical characters, three alphanumerical characters, and three numerical characters. Similarly

PICTURE IS 999.99

represents a number composed of three numerical characters to the left, two to the right, of the decimal point.

The **CLASS** clause indicates the type of data being described and can be **ALPHABETIC**, **NUMERIC**, or **ALPHANUMERIC**. It is written, for example, as

CLASS IS NUMERIC

If the **PICTURE** is given, then the **CLASS** is unnecessary. The **CLASS** description will extend over all higher-numbered associated levels. Thus, if for **DATE** we have **CLASS IS NUMERIC**, then we need not repeat this for the higher-numbered levels of **MONTH**, **DAY**, and **YEAR**. If, on the other hand, the **MONTH** were to be spelled out in the data, then the **CLASS** would have to be given at the **MONTH** level (why?).

If a numerical elementary data-item is to have a sign associated with it, then the word **SIGNED** is included in its description. If the value of some other elementary data-item is to be the sign of the item being described, then **SIGN IS** ⟨data-name⟩ is written. A signed item must be **NUMERIC**, and therefore its class need not be given; also, if a picture that contains an **S** is given, then **SIGNED** is not necessary.

The syntax of the **USAGE** clause is as follows:

⟨usage type⟩ : : = **COMPUTATIONAL-**⟨integer⟩|**DISPLAY-**⟨integer⟩
⟨usage clause⟩ : : = **USAGE IS** ⟨usage type⟩

The purpose of the usage clause is to describe whether the data is to take part in numerical *computations* or is to behave as a word to be *displayed* or "read." The "integer" part is to be a code to indicate the various types of usages, within **COMPUTATIONAL** or **DISPLAY**, that may require special operational handling by the computer. For example, for **COMPUTATIONAL** usage, **1** might represent a binary number, **2** a decimal number, **3** a floating-point number, and so forth. If the usage is **COMPUTATIONAL**, then **CLASS IS NUMERIC** is understood and need not be explicitly stated.

Special-purpose Data Descriptions. There are some important special-purpose data descriptions of COBOL which we shall now consider, namely,

VALUE OCCURS REDEFINES RENAMES

The **VALUE** clause is used to describe the value of a *condition-name*. Recall that a conditional-variable can have several possible option-

values—for instance, the conditional-variable **FACTORY** might have the option-values **A**, **B**, **C**, and **D** if the business in question had four different factories—and that it is frequently convenient to give names to each option-value, called "condition-names." For instance, for the **FACTORY** the name of each of the option-values **A**, **B**, **C**, and **D** might be the name of the city in which each factory is located, such as **NEW-YORK**, **TRENTON**, **WILMINGTON**, and **BALTIMORE**. In such cases we may say that the **VALUE** of New York **IS "A"**, the **VALUE** of Trenton **IS "B"**, and so forth, where quotes (", ") are used around the "value-literal." Of course, the conditional-variable and its possible condition-names and their respective values must be stated in the record description. This is done through the following syntax:

⟨condition-value entry⟩ : : = **88** ⟨condition-name⟩**, VALUE IS "**⟨literal⟩**".**
⟨condition-value list⟩ : : = ⟨condition-value entry⟩|
⟨condition-value list⟩
⟨condition-value entry⟩

⟨condition description⟩ : : = ⟨level number⟩ ⟨condition-variable⟩ ⟨data description⟩
⟨condition-value list⟩

For example, suppose that **FACTORY** is a level-2 item; then for the condition description we would have

2 FACTORY; SIZE IS 1 CHARACTER; CLASS IS ALPHANUMERIC.
88 NEW-YORK, VALUE IS "A".
88 TRENTON, VALUE IS "B".
88 WILMINGTON, VALUE IS "C".
88 BALTIMORE, VALUE IS "D".

Note that the **SIZE** is given as the size of the *value* of the condition variable, that is, the number of characters that a single value requires. The "level number" **88** is reserved solely for the purposes of displaying alternative condition-names and their associated values and has no true level-number meaning.

The **OCCURS** clause is used for defining data tables or arrays. The data-names of the values in the table or array are subscripted variables, and the subscripts specify exactly which values of the table or array of the data-name are desired (like matrices). The **OCCURS** clause is written as

OCCURS ⟨integer⟩ **TIMES**

and can *never* be associated with a record-name, i.e., with a level-**1** data-item description. For example, consider the description of a table of prices of shoes from three different manufacturers, each manufacturer making three different women's shoe lasts and two different men's shoe

lasts, where each of the women's lasts comes in five different colors and each of the men's lasts comes in two different colors. Figure 7-9 is a picture of the table. The table description is as follows:

```
1 TABLE-SHOE-PRICES; SIZE 285 CHARACTERS.
    2 SHOE-MNFR; SIZE IS 95 CHARACTERS;
      CLASS IS NUMERIC; OCCURS 3 TIMES.
        3 WOMENS; SIZE IS 25 CHARACTERS; OCCURS 3 TIMES.
            4 PRICE; PICTURE IS 99.99; OCCURS 5 TIMES.
        3 MENS; SIZE IS 10 CHARACTERS; OCCURS 2 TIMES.
            4 PRICE; PICTURE IS 99.99; OCCURS 2 TIMES.
```

To obtain a value from a table in which the **OCCURS** clause has been used, subscripted variables are used. The subscript **1** refers to the first element of the **OCCURS** list, **2** refers to the second element of the

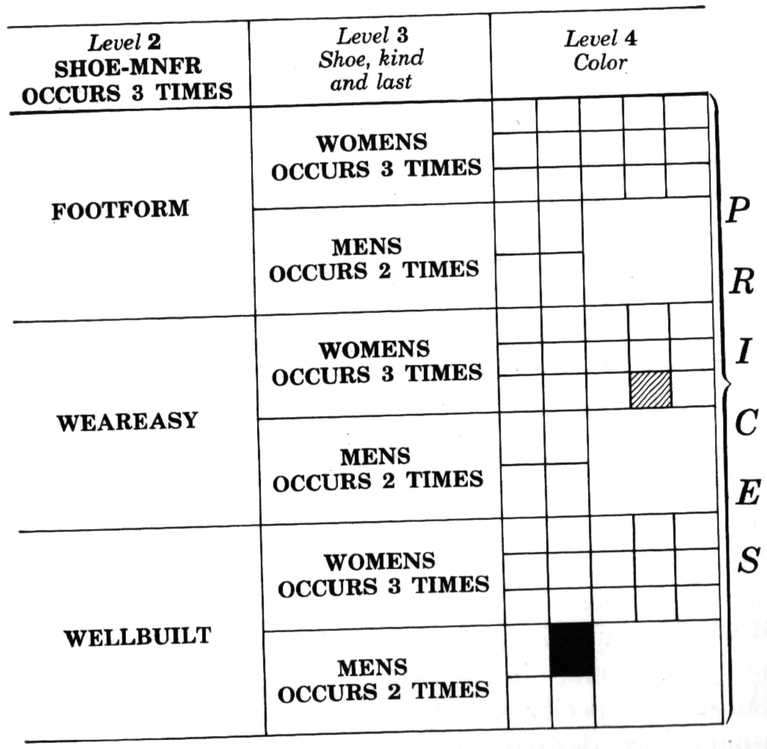


FIG. 7-9. Table of shoe prices according to manufacturer, kind, last, and color. The squares are the places for the prices: the shaded box, for example, contains the price for **WEAREASY WOMENS** shoes, their third last and fourth color; the solid square contains the price for **WELLBUILT MENS** shoes, their first last and second color (see text).

OCCURS list, etc. When more than one subscript is used, the entire data-name that is subscripted may be composed of successive level data-

names, as described above. For example,

PRICE IN WOMENS IN SHOE-MNFR (4, 3, 2)

would represent the price of the shoes of the *fourth* color of the *third* type of women's last of the *second* manufacturer (**WEAREASY**)—see the crosshatched square of Fig. 7-9. Note that the subscripts correspond to the occurrences in the successive component data-names. As another example,

PRICE IN MENS IN SHOE-MNFR (2, 1, 3)

would represent the price of the shoes of the *second* color, the *first* men's last, and the *third* manufacturer—see the solid square in Fig. 7-9. A data-name can appear as a subscript, provided it has an integral value; then the data-name stands for its value in the subscript list. For example,

PRICE IN WOMENS IN SHOE-MNFR (BLUE, 3, WEAREASY)

would represent the crosshatched square in Fig. 7-9, provided that somewhere in the data description there appear the items

```
88 BLUE, VALUE IS 4.
```

and

```
88 WEAREASY, VALUE IS 2.
```

When no ambiguity will result, the data-name need not be composed. For example, consider the following description of a matrix with six columns and three rows:

```
1 MATRIX.
    2 COLUMN; OCCURS 6 TIMES.
        3 ROW-ELEMENT; OCCURS 3 TIMES.
```

The actual matrix would be

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \end{pmatrix}$$

To refer to, say, the *fourth* element in the *second* row, we would write

ROW-ELEMENT (2, 4)

where **2** represents the second **ROW-ELEMENT** and **4** the fourth **COLUMN**, this latter being understood.

It is frequently desirable to have alternative data-names as well as structures for the same collection of elementary items in a record. In such a case, the **REDEFINES** clause is used as follows to indicate the

alternative desired:

⟨level-number⟩ ⟨data-name⟩ **REDEFINES** ⟨data-name⟩; ⟨data description⟩

The *alternative* data-name is to the *left*, the *original* to the *right*, of **REDEFINES**; the new description then follows the semicolon. Both data-names, and both the original and new data descriptions, can then be used in the same program alternatively. For example, suppose that the original description were

```
3 WOMENS-SHOE-COLOR; SIZE IS 25 CHARACTERS;
  USAGE IS COMPUTATIONAL.
    4 BLUE-PRICE; PICTURE IS 99.99.
    4 GREEN-PRICE; PICTURE IS 99.99.
    4 YELLOW-PRICE; PICTURE IS 99.99.
    4 PURPLE-PRICE; PICTURE IS 99.99.
    4 ORANGE-PRICE; PICTURE IS 99.99.
```

The prices must then be referenced by naming the colors explicitly. If it were also desired, as an alternative, to get the price through a subscripted variable, then we would write

```
3 WOMENS-SHOE REDEFINES WOMENS-SHOE-COLOR;
  SIZE IS 25 CHARACTERS.
    4 COLOR-PRICE; PICTURE IS 99.99;
      OCCURS 5 TIMES.
```

Note that the data being described is the *same in both cases*, namely, a list of 5 prices of five characters each. Note that the range of the data encompassed by a **REDEFINES** clause is determined by the level number of the original data-name; the new data-name must, of course, have this same level number.

There is another interpretation of the **REDEFINES** clause: instead of considering it as effecting a redefinition of the *same data*, it can be considered as effecting the redefinition of the *same storage area* or part of the high-speed memory. This latter interpretation becomes useful when there are involved in a program two different types of records that are never utilized at the same time. If these records are of approximately the same length, then by means of the **REDEFINES** clause the same area of the computer's memory can be used to store either record. In this way the computer's memory can be more efficiently utilized. For otherwise two separate areas of the memory would have to be reserved for the records, even though both types of records would never be required in the memory at the same time.

The **RENAMES** clause permits alternative or overlapping groupings or classifications of the elementary data-items of a record. The form

of the **RENAMES** clause is

66 ⟨data-name⟩ **RENAMES** ⟨data-name⟩

where the level number **66** is used only to identify a **RENAMES** clause and the new or alternative data-name is to the *left* of **RENAMES**. These clauses should directly follow the record description in which the renamed elementary data-items occur. For example, suppose there are five cost types included in a record and they are classified as costs for project 1 or project 2; suppose an alternative classification of these cost types into "direct" and "overhead" costs is desired, as indicated in the following table:

	Cost type				
	A	*B*	*C*	*D*	*E*
Project 1 cost.....	1	2			
Project 2 cost.....			1	2	3
Direct cost.....	1		2	3	
Overhead cost.....		1			2

The two classifications would appear as the following record descriptions:

```
2 PROJECT-1; . . . .          2 DIRECT; . . . .
    3 COST-1; . . . .             3 COST-1; . . . .
    3 COST-2; . . . .             3 COST-2; . . . .
2 PROJECT-2; . . . .              3 COST-3; . . . .
    3 COST-1; . . . .         2 OVERHEAD; . . . .
    3 COST-2; . . . .             3 COST-1; . . . .
    3 COST-3; . . . .             3 COST-2; . . . .
```

The left-hand record description is the original record, and it is desired to **RENAME** the elementary data-items so that the right-hand record description will also apply to the data. According to the above table, the following **RENAME** clauses will do the job:

```
66 COST-1 IN DIRECT RENAMES COST-1 IN PROJECT-1.
66 COST-2 IN DIRECT RENAMES COST-1 IN PROJECT-2.
66 COST-3 IN DIRECT RENAMES COST-2 IN PROJECT-2.
66 COST-1 IN OVERHEAD RENAMES COST-2 IN PROJECT-1.
66 COST-2 IN OVERHEAD RENAMES COST-3 IN PROJECT-2.
```

Of course the right-hand record never appears literally, but is implied by the level-**66** entries. The difference between the **REDEFINES** and

the **RENAMES** clauses is that the former redefines the same *areas* or locations in the computer memory regardless of the data stored there, whereas the latter renames the stored *data* itself, effectively (but not actually) introducing a different data-storage organization.

The Working-storage Data Description. So far we have been considering how to describe the record types that are contained in the files. These record descriptions tell the computer (1) the names of the elementary data-items as well as (2) just what area or part of the high-speed memory to reserve for each of the record types to be read into or out of the high-speed memory (from or onto the file tapes). In addition to these input-output file areas, there must be space reserved in the high-speed memory for temporaries to store intermediate results and for constants and initial conditions. Again, the computer must be told the names of these elementary data-items to be stored, and how much space to reserve for them; this is accomplished through the *working-storage data description.*

The working-storage data description includes the working-storage record descriptions, which are formed in exactly the same manner as are the record descriptions discussed above and hence need no further discussion. As in the case of all records, the data-names must be unique when properly compounded, as described above. However, it is frequently desired that space be reserved for data that does not fit into part of a record, such as special temporary locations, initial values, and constants. Descriptions of such data, called *noncontiguous* working-storage data descriptions, are given separate from those of the working-storage records and are *identified by the level number* **77**. They must contain the **SIZE** or **PICTURE** of the data and the **CLASS** and/or **USAGE**, in the manner discussed above. They can also contain any of the other data-description clauses above.

In the working-storage data description, the clause "**VALUE IS . . .**" can be used to give the *initial value* of a data-name. For example, if a sum is being formed, we might have

77 SUM; SIZE IS 7 DIGITS;
CLASS IS NUMERIC; VALUE IS 0000000.

Similarly, a **VALUE IS** clause can also be used for an initial value within a record. For *constants* the same form can be used, except that, of course, the **VALUE IS** clause is required, in order to give the value of the constant: for example,

77 INTEREST-RATE; SIZE IS 2 DIGITS;
CLASS IS NUMERIC; VALUE IS 06.

If the constants bear relationships to one another, then the constants can be written as a record description. Here each of the elementary data-item descriptions must include a **VALUE IS** clause. However, since each elementary data-item is supposed to have only one (constant) value, *no* level-**88** entries can be written (why?).

Tables of constants are handled in a very special way. The table is first described by means of a record description. The values of the entries are given by the **VALUE IS** clause; then this record can be **REDEFINED**, with the **OCCURS** clause in the redefinition. In this way subscripted variables can reference the table of constants (see Exercises 7-37 through 7-40). Here level-**88** entries *can* be written.

7-5. File and Environment Descriptions

The File Description. As we observed above, the data records are generally viewed as being stored on magnetic tape, sequentially. A collection of records is called a *file*, and we have assumed, for the sake of simplicity, that a file contains only a single type of record and that only one file appears on each tape. The data-record description, the topic of the previous section, was needed to describe the organization of the data in the high-speed memory, reserving the appropriate area in the memory for the records and working storage, as well as to "assign" data-names to the various data "locations" in the high-speed memory. The file descriptions, on the other hand, are necessary (1) to detail the *organization* of the records in the magnetic-tape files and identify the files, i.e., to give the number of records in a file and the number of characters in a record, (2) to give the *names* of the record-types of the file, and (3) to give the formats of additional labels that may have been put on the beginning or end of the file tapes for checking and handling purposes. Whereas the record description was generic in character, describing the format of a record-type whose file may contain many thousands of such records, there is only one file associated with each file description. The form of the file description is

⟨file description⟩ : : = **FD** ⟨file-name⟩**;** ⟨file data⟩ {label-record / descriptions}

⟨file data⟩ : : = ⟨file organization⟩ ⟨label-record types⟩
⟨data-record names⟩

Here **FD** denotes that what follows is a file description; the file-name is a word; the file data has three parts, which we shall now describe in detail.

The File Data. The "file-organization" part of the file data contains three clauses:

FILE CONTAINS **BLOCK CONTAINS** **RECORD CONTAINS**

The first is written as

FILE CONTAINS ABOUT ⟨integer⟩ **RECORDS;**

where the integer gives the approximate number of records in the particular file in question, which may be useful in the handling of the file tape. Usually a group, or *block*, of computer (machine-language) words at a time is read into (out of) the high-speed memory from (onto) the magnetic tapes. If the number of computer words is w, and if each record will require r such computer words, then the number of records per block is about w/r, taken to the nearest integral value. Thus, instead of one record at a time being read into (or written out of) the high-speed memory as we have previously assumed, in actuality several records at a time are transferred. These records are stored in the computer memory "on the side," and only one record at a time (from each file) is put into the *record area* of that particular file. Hence we can say that effectively one record at a time from each file is handled by the computer. However, for manipulation purposes we must tell the computer that each

BLOCK CONTAINS ⟨integer⟩ **RECORDS;**

Finally, it is helpful in handling the file tapes for the computer to be told in the file description that each

RECORD CONTAINS ⟨integer⟩ **CHARACTERS;**

(even though the record description itself specifies this number).

The data-record-name part of the file data is of the form

⟨record-name list⟩ : : = ⟨record-name⟩|⟨record-name list⟩, ⟨record-name⟩

$$\textbf{DATA RECORDS} \left\{ \begin{matrix} \textbf{IS} \\ \textbf{ARE} \end{matrix} \right\} \langle \text{record-name list} \rangle .$$

where a "record-name" is the data-name of the first level of a data record, i.e., the true record name. We have been making the simplifying assumption that there is only one record-type in a file, in which case only its name would appear in the "record-name list" and the **IS** option would be used. If more than one record-type is in a file, then the names of each of the record-types would appear in the "record-name list" and the **ARE** option would be used.

Label Records. The "label-record descriptions" part of the file description and the "label-record types" part of the file data concern the notion of label records themselves. A label record is a special record that is placed on the file tape at the beginning, and often also at the end, of the file and/or tape (see Fig. 7-10). The constituents of a label record itself depend on what computer is being used, as well as on the particular problem for which the file was prepared. The label records give such things as the date of composition of the file, the date at which the file will be obsolete, the reel number of a file that requires several reels, and

other checking and file- or tape-handling information that may be deemed necessary or helpful. The description of a label record is similar to, and can use the clauses of, the data-record discussed above. As seen from Fig. 7-10, there are four types of label records that may be used on a tape:

⟨label-record type⟩ : : = **BEGINNING-TAPE**|**BEGINNING-FILE**
|**ENDING-FILE**|**ENDING-TAPE**

$$\left\langle \begin{matrix} \text{label-record-} \\ \text{type list} \end{matrix} \right\rangle ::= \left\langle \begin{matrix} \text{label-record} \\ \text{type} \end{matrix} \right\rangle \Bigg| \left\langle \begin{matrix} \text{label-record-} \\ \text{type list} \end{matrix} \right\rangle , \left\langle \begin{matrix} \text{label-record} \\ \text{type} \end{matrix} \right\rangle$$

Hence the "label-record type" part of the file data includes the clause

$$\textbf{LABEL RECORDS} \left\{ \begin{matrix} \textbf{IS} \\ \textbf{ARE} \end{matrix} \right\} \left\{ \begin{matrix} \langle \text{label-record-type list} \rangle \\ \textbf{STANDARD} \\ \textbf{OMITTED} \end{matrix} \right\}$$

The use of the **STANDARD** option implies that the label records adhere in type and in individual format to a standard form set up by computer manufacturer or other implementer. If this "standard" option is used, then the label-record description need not be explicitly included, but is "understood." A standard label-record description follows:

```
1 BEGINNING-TAPE; SIZE IS 25 CHARACTERS;
  CLASS IS ALPHANUMERIC.
    2 IDENTIFICATION; SIZE IS 9 CHARACTERS;
      CLASS IS ALPHANUMERIC.
    2 REEL-NUMBER; SIZE IS 4 DIGITS; CLASS IS NUMERIC.
    2 DATE-WRITTEN; SIZE IS 6 DIGITS; CLASS IS NUMERIC.
        3 MO; SIZE IS 2 DIGITS.
        3 DA; SIZE IS 2 DIGITS.
        3 YR; SIZE IS 2 DIGITS.
    2 PURGE-DATE; SIZE IS 6 DIGITS; CLASS IS NUMERIC.
        3 MO; SIZE IS 2 DIGITS.
        3 DA; SIZE IS 2 DIGITS.
        3 YR; SIZE IS 2 DIGITS.
```

When it is not desired to have label records at all, then the **OMITTED** option is used and the label-record descriptions are omitted. In all other cases, the label-record-type list is given and the label-record descriptions are included.

The "label-record types" part of the file data includes another clause, of the form

$$\textbf{VALUE OF} \left\langle \begin{matrix} \text{data-name in} \\ \text{label record} \end{matrix} \right\rangle \textbf{IS} \left\{ \begin{matrix} \langle \text{data-name in constants} \rangle \\ \langle \text{data-name in data-record} \rangle \ \textbf{HASHED} \end{matrix} \right\}$$

The purpose of this clause is to serve as a check or identification, and it can be used in either of two possible ways. In the first, the label record on the tape contains a certain constant or word, as the "contents" of the data-name following **VALUE OF**; this constant or word is then compared with a constant, whose data-name follows the **IS**, in the constants section of the working storage: if the comparison is successful, then the identification of the file has been confirmed; otherwise an error is indicated by the computer. In the second way, the label record on the tape again

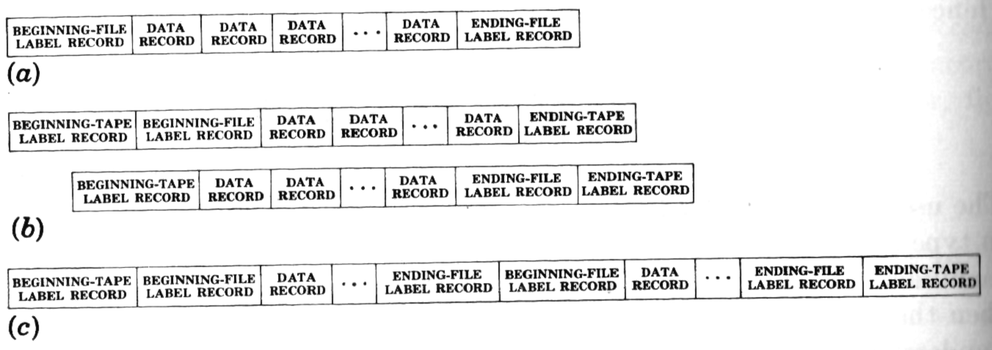


FIG. 7-10. (*a*) Illustration of records in a file that takes up a single tape reel; (*b*) illustration of records in a file that requires two (or more) tape reels; (*c*) illustration of records in files where more than one file appears on a tape.

contains a certain constant or word; this time, however, it is not compared with the "contents" of the data-name in the data record (given after **IS**), but rather with the *sum* of the contents of the data-record data-names of each record of the file—i.e., the summation is taken over all the records of the file under consideration. This is the meaning of the word **HASHED** that appears in this second option of the clause. Then, as each data-record is read into (out of) the high-speed memory, the total sum is formed and at the proper time (i.e., when all the data-records have been read) this total is compared with the label-record data-name for the check. In this way, an omitted data-record or other such error can be detected.

The "label-record descriptions" give the *format* of each label record associated with the file under consideration, and these records appear after the file data. (The label records themselves appear on the file tape as shown in Fig. 7-10.) The following paragraph is an example of a complete file description, which also includes (nonstandard) label-record descriptions:

```
FD PAYROLL-DATA; FILE CONTAINS ABOUT 1000 RECORDS;
BLOCK CONTAINS 10 RECORDS; RECORD CONTAINS 70 CHAR
```



```
ACTERS; LABEL RECORDS ARE BEGINNING-FILE, ENDING-
FILE; VALUE OF ID IS FILE-NAME; DATA RECORDS IS MASTER.
    1 BEGINNING-FILE; SIZE IS 19 CHARACTERS;
      CLASS IS ALPHANUMERIC.
        2 ID; SIZE IS 3 CHARACTERS;
          CLASS IS ALPHANUMERIC.
        2 REEL-NUMBER; SIZE IS 4 DIGITS;
          CLASS IS NUMERIC.
        2 JOBCLASSIFICATION; SIZE IS 6 CHARACTERS;
          CLASS IS ALPHANUMERIC.
        2 LASTUPDATING; SIZE IS 6 DIGITS.
            3 MO; SIZE IS 2 DIGITS.
            3 DA; SIZE IS 2 DIGITS.
            3 YR; SIZE IS 2 DIGITS.
    1 ENDING-FILE; SIZE IS 12 DIGITS; CLASS IS NUMERIC.
        2 RECORD COUNT; SIZE IS 3 DIGITS.
        2 PAYROLL TOTAL; SIZE IS 9 DIGITS.
```

The Environment Description.† In the previous section, we considered how the data that enters into a problem can be described in terms of record formats; in this section, we have considered how a collection of records could be organized in a file, and how such a file can be described. In the present paragraph, we shall describe how the files themselves can be related to the equipment available on the particular computer being used—i.e., the *environment description.* As we noted above, the simplest concept is to have one file on a single magnetic-tape unit: an input file would be contributing data (records) to the high-speed memory for processing, and an output file would be receiving from the high-speed memory information which results from the processing. Of course, the input file could be on punched cards instead of magnetic tapes, the output file could be the printed results from a high-speed printer, and so forth. Hence, when a problem is being run, each piece of input or output equipment (hardware) which is being used must be assigned to a particular file.

The assignment of files to the input and output units (hardware) is accomplished through the *input-output section,* which has the form

⟨hardware list⟩ : : = ⟨hardware-name⟩|⟨hardware list⟩, ⟨hardware-name⟩
⟨assignment⟩ : : = **SELECT** ⟨file name⟩, **ASSIGN TO** ⟨hardware list⟩.

† In this environment description we have chosen to omit many of the alternative possibilities provided for in COBOL; this has been done for ease in understanding of the concepts and to eliminate unnecessary complications for the student in his first exposure to the language.

⟨assignment list⟩ : : = ⟨assignment⟩|
⟨assignment list⟩
⟨assignment⟩
⟨input-output section⟩ : : = **FILE CONTROL.**
⟨assignment-list⟩

For example, suppose that our computer has four magnetic-tape units and one printer, named (hardware-names) **MTU1**, **MTU2**, **MTU3**, **MTU4**, and **PRINTER**. Suppose also that, for a particular problem, we intended to put the file named **MASTER** on tape **MTU1**, the file named **TRANSACTION** on tapes **MTU2** and **MTU3** (that is, this file requires two tapes), the file named **NEW-MASTER** on tape **MTU4**, and the file named **REPORT** on the **PRINTER**. The input-output section would then appear as

FILE CONTROL.
SELECT MASTER, ASSIGN TO MTU1.
SELECT TRANSACTION, ASSIGN TO MTU2, MTU3.
SELECT NEW-MASTER, ASSIGN TO MTU4.
SELECT REPORT, ASSIGN TO PRINTER.

In addition, the environment description should include some description of the computer itself, of that part of the high-speed memory of the computer which is to be reserved for the program in question, and of those input-output units (hardware) which are to be utilized by the computer. This information is specified in the *configuration section*, which has the following form:

⟨memory segment⟩ : : = ⟨literal⟩ **THRU** ⟨literal⟩
⟨memory-segment list⟩ : : = **ADDRESS** ⟨memory segment⟩ | ⟨memory-segment list⟩, ⟨memory segment⟩
⟨configuration section⟩ : : = **OBJECT-COMPUTER.** ⟨computer name⟩
ASSIGN OBJECT-PROGRAM TO ⟨hardware name⟩.
MEMORY SIZE ⟨memory-segment list⟩.
⟨hardware list⟩.

The term **OBJECT-COMPUTER** refers to the computer being used to run the program, for example, the IBM 7090, RCA 501, Honeywell 800, etc. The **OBJECT-PROGRAM** is the program, written in the COBOL language, which is to be run; this program itself must be stored on an input unit. The configuration section of the program must be read into the computer from the input unit before the computer will know where the rest of the program is located. The **MEMORY SIZE** clause specifies the segments of the high-speed memory that will be utilized in the program, and the hardware list is the list of those input-output units that will be used in the running of the program. These should, of course, agree with input-output (**FILE CONTROL**) section. As an example, suppose we are to run a program on a computer called the **PEDAGAC**, that the object-program is on tape **MTU5**, that memory addresses 000 through 400 and 600 through 777 are to be used, and that input-output units **MTU1**, **MTU2**, **MTU3**, **MTU4**, and the **PRINTER** are to be used. The configuration section would then be

OBJECT-COMPUTER. PEDAGAC
ASSIGN OBJECT-PROGRAM TO MTU5.
MEMORY SIZE ADDRESS 000 THRU 400, 600 THRU 777.
MTU1, MTU2, MTU3, MTU4, PRINTER.

With a slight stretch of the imagination, we can consider the name of the writer or author (programmer) of a program, as well as other information concerning the writing of the program, as part of the program's environment. This *program identification* section has the following form:

⟨program identification⟩ : : = **PROGRAM ID.** ⟨name⟩|
⟨program identification⟩
AUTHOR. ⟨name⟩|
⟨program identification⟩
DATE WRITTEN. ⟨date⟩|
⟨program identification⟩
REMARKS. ⟨discussion⟩.

For example,

PROGRAM ID. PAYROLL
AUTHOR. ROBERT S LEDLEY
DATE-WRITTEN. JULY 22, 1961
REMARKS. PAYROLL-DATA-FILE AND TOTAL-MASTER-FILE NEEDED AS INPUTS FOR THIS PROGRAM.

In conclusion, the Data Division of our simplified version of COBOL is organized as shown in Fig. 7-11. In the actual COBOL, the file-control and object-computer parts are included in a separate *Environment Division*, the program identification is included in a separate *Identification Division*, and the *Data Division* includes only what we have called the data description.

7-6. Data Handling by COBOL

Orientation. So far we have considered (1) the *operational statements* of COBOL, which process the data by computing functions and carrying out the actual numerical calculations (Sec. 7-2), and (2) the *data-descrip-*

tion clauses of COBOL, which define the data (as previously organized) on tape and reserve the proper high-speed-memory areas for input and output files and working storage (Secs. 7-4 and 7-5). There remain three further topics necessary to complete the discussion of COBOL. The first is the *data-handling statements* of COBOL, used to move or manipulate the data—in a manner analogous to the copying of entries into different

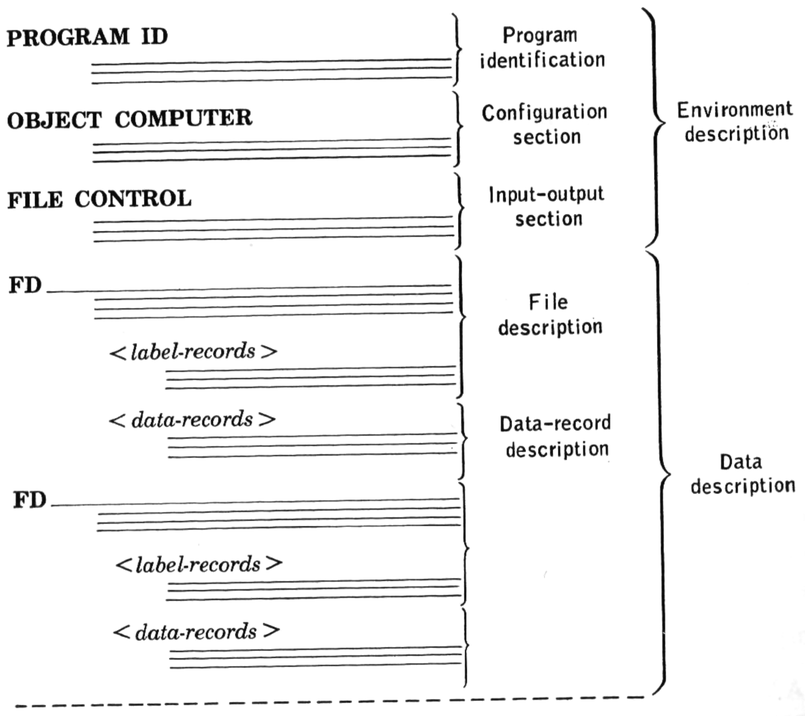


FIG. 7-11. Data Division of COBOL sharing only two file descriptions and associated record descriptions; the dashed line indicates that there are more of these if more than two files are used.

columns which is so characteristic of bookkeeping; the second is the *data-editing clauses* of COBOL, used to define the proper or desired form of the data for the printing of reports, fiscal summaries, check information, etc.; and the third is the *file- and record-handling statements* that are used to sequence the records of a file tape properly for processing. These we shall now consider individually.

Data-handling Statements. The data-handling statements are

MOVE and **MOVE CORRESPONDING**

The **MOVE** verb is used to transfer data from an input-file area to a working-storage or output-file area or from a working-storage area to an

output-file area, etc. The syntax of the **MOVE** statement is as follows:

⟨sending field⟩ : : = ⟨data-name⟩
⟨receiving-field list⟩ : : = ⟨data-name⟩|⟨receiving-field list⟩**,** ⟨data-name⟩
⟨move statement⟩ : : = **MOVE** ⟨sending-field⟩ **TO** ⟨receiving-field list⟩

The **MOVE** statement is analogous to a transfer instruction except that the sending field may be moved (transferred) into each of several different "locations" at once (i.e., each of the "locations" of the receiving-field list). For example,

MOVE BADGE-NO IN MASTER TO BADGE-NO IN T-MASTER,
BADGE-NO IN N-MASTER.

This statement would move the elementary data-item with data-name **BADGE-NO** of the **MASTER** record of, say, an input file into the analogous data-name of the working-storage **T-MASTER** record and output-file **N-MASTER** record.

The **MOVE CORRESPONDING** verb is used to move a group of elementary data-items, all at one time, into a corresponding group of new elementary data-item locations. The form of this variation of the **MOVE** statement is

MOVE CORRESPONDING ⟨sending area⟩ **TO** ⟨receiving-area list⟩

where ⟨sending area⟩ : : = ⟨partial data-name⟩
⟨receiving-area list⟩ : : = ⟨partial-data-name list⟩

Here by "partial data-name" we mean a level name under which level higher levels exist. Of course the sending and receiving areas must include corresponding higher associated levels. For example, consider the two records:

```
1 MASTER; SIZE IS 6 DIGITS.      1 N-MASTER; SIZE IS 6 DIGITS.
  2 DATE; SIZE IS 6 DIGITS.        2 DATE; SIZE IS 6 DIGITS.
    3 MO; SIZE IS 2 DIGITS.          3 MO; SIZE IS 2 DIGITS.
    3 DA; SIZE IS 2 DIGITS.          3 DA; SIZE IS 2 DIGITS.
    3 YR; SIZE IS 2 DIGITS.          3 YR; SIZE IS 2 DIGITS.
```

and suppose we wished to transfer the **DATE** from the **MASTER** record to the **N-MASTER** record. We would write

```
MOVE CORRESPONDING DATE IN MASTER
                   TO DATE IN N-MASTER.
```

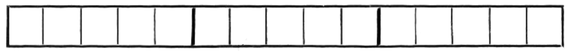
If the data descriptions of the sending and receiving fields or areas are not the same, then the *data-items are moved* (by the **MOVE** verb) *in conformity with the description of the receiving field or area.*

Data-editing Clauses. When business, planning, medical, and other such data are handled, which characteristically require listings, summaries, totals, breakdowns, and so forth, as the final data-processing output, the format of the final print-out is of the utmost importance. The purpose of the data-editing clauses of COBOL is to control and process such formats.

In discussing these clauses, concrete, physical pictures of the location of *each character* of the data must be kept in mind. In a record description, the number of characters in an elementary data-item is specified by either the **SIZE** or the **PICTURE** clause: thus a particular *field* in the computer memory is specified. For example, suppose we specify that the elementary data-item **NAME** is to have 15 characters; or the picture may be given as

XXXXXXXXXXXXXXX

Then the field reserved can be diagramed as



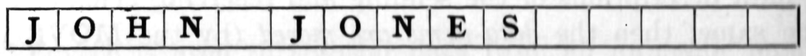
If the name is, say, **JOHN JONES**, which is smaller than the reserved field, the question arises of where it will be located in the field. Again, if data is being moved from one field to another by means of the **MOVE** verb, similar problems arise when the sending and receiving fields are of different sizes.

When data is printed out, it is often desirable to have, for example, the $ to the left of the leftmost digit, as in $5.95 or $25.95, etc. Clearly the actual computations cannot be performed with the dollar sign so placed, especially where it may not be known beforehand how many digits there will be in, say, a final total. Here, as in many other cases, certain symbols must be appropriately placed in the final results, so that the print-out will appear in the proper report form.

The data-editing clauses utilized when such problems arise are

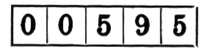
JUSTIFIED **POINT LOCATION** *Editing*

The **JUSTIFIED** clause is used when nonstandard positioning of an elementary data-item is desired in a field that is set up for more characters than the data-item contains. The *standard positioning* is as follows: For alphabetical or alphanumerical data, the data is positioned flush with the *left* end of the field, with blanks filling in any right-hand spaces. For example, in a **NAME** field of 15 characters, the standard positioning of **JOHN JONES** would be



Numerical data for which the positioning of the decimal point has been

specified (see below) must of course be recorded in its field properly positioned with respect to this point position. Other numerical data is positioned flush with the *right* end of the field, with zeros filling in any left-hand spaces. For example, in a field of five characters, the *standard position* of 595 is



When such standard positioning of data is *not* desired, the **JUSTIFIED** clause can be used in the data description, in the following alternative forms:

JUSTIFIED { **LEFT** / **RIGHT** }

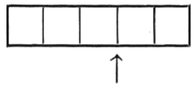
The positioning of a decimal (or binary) point *in a number* is given by the **PICTURE** clause described above; the positioning of the decimal (or binary) point *in a field* in which a number is to be placed is specified by the **POINT LOCATION** clause:

POINT LOCATION IS LEFT ⟨integer⟩ { **PLACES** / **BITS** }

Here "integer" gives the number of positions to the *left* of the *rightmost* (least significant) position of the field, at which the decimal (or binary) point location is to be *assumed.* The decimal point does not actually appear in the field, because of the point-location clause. For example, for a field for which

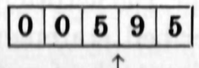
SIZE IS 5 DIGITS; POINT LOCATION IS LEFT 2 PLACES

we have

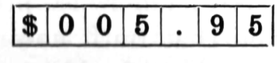


To obtain an actual decimal (or binary) point character, the **PICTURE** clause must be utilized. If data is moved (by means of a **MOVE** statement) from a sending field with a **POINT LOCATION** clause to a receiving field with a **PICTURE** clause, then the decimal (or binary) points are made compatible. For example,

Sending field — **POINT LOCATION IS LEFT 2 PLACES**



⇒ *Receiving field* — **PICTURE IS $999.99**



The *Editing* clauses enable three kinds of "variable" editing options to be applied to numbers. These are written as

ZERO SUPPRESS
CHECK PROTECT
FLOAT DOLLAR SIGN

An editing option is applied to a number from left to right and suppresses (or changes) the leading zeros until a nonzero digit or the decimal point (actual or assumed) is encountered; the option is automatically applied to a number as it is moved (by a **MOVE** verb) into a field or area whose data description includes such an editing option. The effect of each of the editing options is as follows:

1. **ZERO SUPPRESS.** The leading zeros are replaced by blanks. For example,

Sending field		*Receiving field* ZERO SUPPRESS
0 0 0 5 9 5	⇒	5 9 5

2. **CHECK PROTECT.** The leading zeros are replaced by asterisks. For example,

Sending field POINT LOCATION IS LEFT 2 PLACES		*Receiving field* PICTURE IS $999.99; CHECK PROTECT
0 0 5 9 5	⇒	$ * * 5 . 9 5

3. **FLOAT DOLLAR SIGN.** The rightmost zero suppressed will be replaced by the dollar sign. For example,

Sending field POINT LOCATION IS LEFT 2 PLACES		*Receiving field* PICTURE IS $999.99; FLOAT DOLLAR SIGN
0 0 5 9 5	⇒	$ 5 . 9 5

Note that, as illustrated by the last few examples, the **PICTURE** clause itself can accomplish some editing. The editing clauses then lend further flexibility to the editing processes.

File- and Record-handling Statements. Figure 7-5 described the process by which successive records were transferred from the file tapes into the high-speed memory for processing. However, the astute reader will have observed that until now we have not described statements of COBOL which accomplish this task, or the companion task of putting processed information onto file tapes, or "onto" high-speed printers. We shall

describe these input-output statements:

OPEN CLOSE READ WRITE

Prior to the initial use of an input or output file, an **OPEN** statement must be applied to the file-name. The execution of an **OPEN** statement establishes an input- or output-file storage area and performs appropriate checks on the file label records. Similarly, when no further use is required of an input or output file, a **CLOSE** statement must be applied to the file-name. The **CLOSE** statement releases the input- or output-file area for other use, performs the appropriate closing conventions that may involve the label-records of the file, and usually rewinds the file-tape reels. A file may be closed only once for each time it is opened; an opened file cannot be opened again until a **CLOSE** statement has first been applied to its file-name. The formats of the **OPEN** and **CLOSE** statements are as follows:

⟨file-name list⟩ : : = ⟨file-name⟩|⟨file-name list⟩, ⟨file-name⟩
⟨in-out file-name list⟩ : : = **INPUT** ⟨file-name list⟩
|**OUTPUT** ⟨file-name list⟩
⟨open statement⟩ : : = **OPEN** ⟨in-out file-name list⟩
⟨rewind options⟩ : : = **NO REWIND**|**LOCK**
⟨file-name options⟩ : : = ⟨file-name⟩|⟨file-name⟩ **WITH** ⟨rewind options⟩
⟨file-name-options list⟩ : : = ⟨file-name options⟩
|⟨file-name-options list⟩, ⟨file-name options⟩
⟨close statement⟩ : : = **CLOSE** ⟨file-name-options list⟩

The execution of the "open statement" will apply all the opening procedures to each file named in the file-name list. The execution of the "close statement" will apply all the closing procedures to each file named in the file-name-options list. If no "rewind options" are associated with a file-name, then the file is rewound automatically; if the **NO REWIND** option is used, the tape is left in its last position before the close statement was executed; if the **LOCK** option is used, the tape is rewound but special precautions are taken so that the tape cannot be read from or written upon again.

After an "open statement" is applied to a file, it is called *open* until a close statement is applied to it, after which it is called *closed.* (Also, before the initial "open statement" has been executed, the file is called *closed.*) The **READ** and **WRITE** statements can be applied only to *open files,* for only these files have input or output storage areas reserved in the high-speed memory for their records. Each time a **READ** statement is applied to an input-file-name, one record is (effectively) read from its tape into the input storage area reserved for that file, replacing the previous contents of this storage area. Each time a **WRITE** state-

ment is applied to a record name, this record is written onto the appropriate output-file tape. In this way a COBOL program can be made successively to process one record of a file at a time. A typical sequence of events might be as in Fig. 7-12.

The **READ** statement has the following format:

READ ⟨file-name⟩ **RECORD; AT END** ⟨nonconditional statement⟩.

When a file consists of more than one type of record, these records automatically share the same storage area, sequentially. This is equivalent to saying that there exists an implicit redefinition of the area, and that

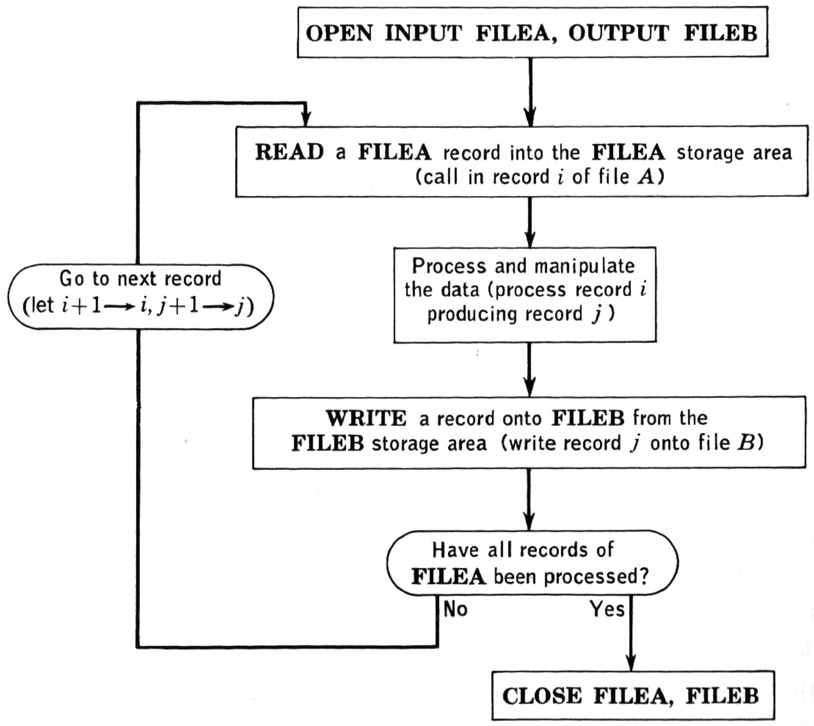


FIG. 7-12. Typical loop for sequencing the processing of the records of a file.

only the information which is present in the current record, just "read in" by means of the most recent **READ** instruction, is available. No reference in the processing of the record (by the procedure division) can be made to information which is not actually present in the current record. When all the records have been **READ** from a file, the end-of-file label record is sensed and the "nonconditional statement" following **AT END** is executed. For instance, we might have

READ FILEA RECORD; AT END GO TO LABELA.

Here, after the last record of **FILEA** has been processed, the program

will jump to the subroutine labeled **LABELA**, which provides the exit for the processing loop.

The **WRITE** statement has this format:

⟨from clause⟩ ::= **FROM** ⟨record-name⟩

$$\begin{Bmatrix}\text{advancing}\\ \text{clause}\end{Bmatrix} ::= \begin{Bmatrix}\textbf{AFTER}\\ \textbf{BEFORE}\end{Bmatrix} \textbf{ADVANCING} \begin{Bmatrix}\langle\text{data-name}\rangle\ \textbf{LINES}\\ \langle\text{integer}\rangle\ \textbf{LINES}\end{Bmatrix}$$

$$\begin{Bmatrix}\text{write}\\ \text{statement}\end{Bmatrix} ::= \textbf{WRITE}\ \langle\text{record-name}\rangle\ [\langle\text{from clause}\rangle]\ \left[\begin{Bmatrix}\text{advancing}\\ \text{clause}\end{Bmatrix}\right]$$

The "record-name" following **WRITE** must be the name of a record in the output-file storage area, which thus specifies which file is to be written upon. This record is then written out onto its associated file each time the **WRITE** statement is executed. The [square] brackets indicate two possible options, either *or both* of which may be included or omitted. If the record to be written out is not in an output-file area, but is in an input-file area or working-storage area, then the "from clause" is used, with the name of the desired record in these other areas included in the "from clause." The effect of using the "from clause" is to **MOVE** the record named in the clause automatically to the record named after **WRITE**, and then to write this onto the associated file. The "advancing clause" is used only when the output file is actually to be a print-out on a high-speed printer.

A Complete Example. At the end of this section we present a complete example of a simplified COBOL program to do a monthly payroll. Three files are involved, the initial payroll information file, called **PAYROLL-INFO-FILE**, the updated payroll file to be used the following month, called **NEW-PAYROLL-INFO-FILE**, and the paycheck data file, called **PAYCHECK-FILE**. For simplicity it is assumed that each employee is paid on a regular monthly salary rate. Each month the Social Security and Federal income-tax withholdings are deducted from this monthly salary rate, according to the following formulas:

1. Monthly Social Security is 0.031 × salary rate, until the accumulated payment has reached $148.80, after which no further payments (deductions) are made.
2. Monthly income-tax withholding = 0.18 × (salary rate − N × 56), where N is the number of dependents claimed. Of course the tax withholding cannot be negative.

The amount of the monthly paycheck is equal to the salary rate less these two deductions. In addition, the new payroll file must be updated with the total income tax withheld to date and the total Social Security withheld to date, as well as the total gross salary earned to date. The student should study the example carefully to see how each of these items is accomplished.

In a more realistic payroll problem, the occurrence of a new employee, the dropping of an employee, the changing of the salary rate, or the number of dependents should also be taken into account (see Exercise 7-42). In addition, hourly wages and part-time employment must also be handled (see Exercise 7-44). Other withholding items, such as state income tax, retirement and insurance, regular savings or bond buying, and so forth, must often be considered (see Exercises 7-43 and 7-45). Finally, the employer's monthly deposit of withholdings must be computed, and the quarterly form 941, annual forms 940 and W-3, and the W-2 form for each employee must all be computed. Thus it becomes clear that even such an apparently straightforward data-processing problem can become quite complex. The student will gain valuable experience by trying the exercises mentioned.

```
DATA DIVISION.

PROGRAM ID PAYROLL.
    AUTHOR.  ROBERT S. LEDLEY
    DATE WRITTEN.  JULY 22, 1961
    REMARKS.  PAYROLL-INFO-FILE IS INPUT, PAYCHECK-
        FILE AND NEW-PAYROLL-FILE ARE OUTPUTS, NO
        CHANGES IN PAYROLL DATA LIST ARE INCLUDED.

OBJECT COMPUTER.  PEDAGAC
    ASSIGN OBJECT-PROGRAM TO MTU5.
    MEMORY SIZE ADDRESS 000 THRU 1000.
    MTU1, MTU2, PRINTER.

FILE CONTROL.
    SELECT PAYROLL-INFO-FILE, ASSIGN TO MTU1.
    SELECT NEW-PAYROLL-INFO-FILE, ASSIGN TO MTU2.
    SELECT PAYCHECK-FILE, ASSIGN TO PRINTER.

FD PAYROLL-INFO-FILE; FILE CONTAINS ABOUT 1000 RECORDS;
    BLOCK CONTAINS 10 RECORDS; RECORD CONTAINS 50
    CHARACTERS; LABEL RECORDS ARE BEGINNING-FILE,
    ENDING-FILE; VALUE OF ID IS PIF; DATA RECORDS IS
    PAYROLL.
      1 BEGINNING-FILE; SIZE IS 9 CHARACTERS;
        CLASS IS ALPHANUMERIC.
          2 ID; SIZE IS 3 CHARACTERS; CLASS IS ALPHABETIC.
          2 LASTUPDATING; SIZE IS 6 DIGITS.
              3 MO; SIZE IS 2 DIGITS.
              3 DA; SIZE IS 2 DIGITS.
              3 YR; SIZE IS 2 DIGITS.
```



```
      1 ENDING-FILE; SIZE IS 3 DIGITS; CLASS IS NUMERIC.
          2 BLOCK COUNT; SIZE IS 3 DIGITS.
      1 PAYROLL; SIZE IS 50 CHARACTERS;
        CLASS IS ALPHANUMERIC.
          2 NAME; SIZE IS 12 CHARACTERS;
            CLASS IS ALPHABETIC.
              3 LAST; SIZE IS 10 CHARACTERS.
              3 INITIALS; SIZE IS 2 CHARACTERS.
          2 SSNO; SIZE IS 9 DIGITS.
          2 SALARY-RATE; PICTURE IS 9999.99.
          2 EXEMPTIONS; SIZE IS 1 DIGIT.
          2 GROSS-TO-DATE; PICTURE IS 99999.99.
          2 SSWHLD-TO-DATE; PICTURE IS 999.99.
          2 FED-TAX-WHLD-TO-DATE; PICTURE IS 9999.99.

FD PAYCHECK-FILE; FILE CONTAINS ABOUT 1000 RECORDS;
    BLOCK CONTAINS 10 RECORDS; RECORD CONTAINS 42
    CHARACTERS; LABEL RECORDS OMITTED; DATA RECORDS
    IS CHECK.
      1 CHECK; SIZE IS 42 CHARACTERS;
        CLASS IS ALPHANUMERIC.
          2 NAME; SIZE IS 12 CHARACTERS;
            CLASS IS ALPHABETIC.
              3 LAST; SIZE IS 10 CHARACTERS.
              3 INITIALS; SIZE IS 2 CHARACTERS.
          2 GROSS-PAY; PICTURE IS $9999.99.
          2 DEDUCTIONS.
              3 SS; PICTURE IS $999.99; FLOAT DOLLAR SIGN.
              3 FED-TAX; PICTURE IS $999.99;
                FLOAT DOLLAR SIGN.
          2 AMOUNT; PICTURE IS $9999.99; CHECK PROTECT.

FD NEW-PAYROLL-INFO-FILE; FILE CONTAINS ABOUT 1000
    RECORDS; BLOCK CONTAINS 10 RECORDS; RECORD CON
    TAINS 50 CHARACTERS; LABEL RECORDS ARE BEGINNING-
    FILE; ENDING-FILE; VALUE OF ID IS NPF; DATA RECORDS
    IS NEW-PAYROLL.
      1 BEGINNING-FILE; SIZE IS 9 CHARACTERS;
        CLASS IS ALPHANUMERIC.
          2 ID; SIZE IS 3 CHARACTERS; CLASS IS ALPHABETIC.
          2 DATE; SIZE IS 6 DIGITS.
              3 MO; SIZE IS 2 DIGITS.
              3 DA; SIZE IS 2 DIGITS.
              3 YR; SIZE IS 2 DIGITS.
```

```
1 ENDING-FILE; SIZE IS 3 DIGITS; CLASS IS NUMERIC.
    2 BLOCK COUNT; SIZE IS 3 DIGITS.
1 NEW-PAYROLL; SIZE IS 50 CHARACTERS;
  CLASS IS ALPHANUMERIC.
    2 NAME; SIZE IS 12 CHARACTERS;
      CLASS IS ALPHABETIC.
        3 LAST; SIZE IS 10 CHARACTERS.
        3 INITIALS; SIZE IS 2 CHARACTERS.
    2 SSNO; SIZE IS 9 CHARACTERS.
    2 SALARY-RATE; PICTURE IS 9999.99.
    2 EXEMPTIONS; SIZE IS 1 DIGIT.
    2 GROSS-TO-DATE; PICTURE IS 99999.99.
    2 SS WHLD-TO-DATE; PICTURE IS 999.99.
    2 FED-TAX-WHLD-TO-DATE; PICTURE IS 9999.99.

WORKING STORAGE.
  77 SOC-TEMP; PICTURE IS 999.99.
  77 TAX-TEMP; PICTURE IS 9999.99.
  77 SS-RATE; SIZE IS 3 CHARACTERS; CLASS IS NUMERIC;
     POINT LOCATION IS LEFT 3 PLACES; VALUE IS 031.
  77 PIF; SIZE IS 3 CHARACTERS; CLASS IS ALPHABETIC.
  77 NPF; SIZE IS 3 CHARACTERS; CLASS IS ALPHABETIC.
  77 ZERO; PICTURE IS 9999.99; VALUE IS 0000.00.

PROCEDURE DIVISION.

HOUSEKEEPING-OPEN.
    OPEN INPUT PAYROLL-INPUT-FILE, OUTPUT PAYCHECK-
    FILE, NEW-PAYROLL-INFO-FILE. READ PAYROLL-INPUT
    FILE RECORD; AT END GO TO HOUSEKEEPING-CLOSE.†

UPDATING-SSWHLD-AND-FED-TAX.
    COMPUTE SOC-TEMP FROM SS-RATE TIMES SALARY-RATE
    IN PAYROLL. IF SOC-TEMP IS GREATER THAN 148.80
    MINUS SSWHLD-TO-DATE IN PAYROLL THEN MOVE 148.80
    TO SSWHLD-TO-DATE IN NEW-PAYROLL; COMPUTE SS IN
    DEDUCTIONS FROM 148.80 MINUS SSWHLD-TO-DATE IN
    PAYROLL ELSE MOVE SOC-TEMP TO SS IN DEDUCTIONS;
    COMPUTE SSWHLD-TO-DATE IN NEW-PAYROLL FROM
    SOC-TEMP PLUS SSWHLD-TO-DATE IN PAYROLL. COM
    PUTE TAX-TEMP FROM 0.18 × (SALARY-RATE IN PAYROLL
    MINUS EXEMPTIONS TIMES 56.00). IF TAX-TEMP IS
    GREATER THAN ZERO THEN MOVE TAX-TEMP TO FED-
```

† **AT END** clause is required by the syntax, even though the program never returns to this point.


```
    TAX IN DEDUCTIONS; COMPUTE FED-TAX-WHLD-TO-DATE
    IN NEW-PAYROLL FROM TAX-TEMP PLUS FED-TAX-WHLD-
    TO-DATE IN PAYROLL ELSE MOVE ZERO TO FED-TAX IN
    DEDUCTIONS; MOVE FED-TAX-WHLD-TO-DATE IN PAY
    ROLL TO FED-TAX-WHLD-TO-DATE IN NEW-PAYROLL.

GROSS-PAY-AND-AMOUNT.
    COMPUTE GROSS-PAY IN CHECK FROM SALARY-RATE.
    COMPUTE AMOUNT IN CHECK FROM GROSS-PAY MINUS
    SS MINUS FED-TAX. COMPUTE GROSS-TO-DATE IN NEW-
    PAYROLL FROM GROSS-TO-DATE IN PAYROLL PLUS
    SALARY-RATE.

OUTPUT-FILE-MAKEUP.
    MOVE CORRESPONDING NAME IN PAYROLL TO NAME IN
    CHECK, NAME IN NEW-PAYROLL. MOVE SSNO IN PAY
    ROLL TO SSNO IN NEW-PAYROLL. MOVE SALARY-RATE
    IN PAYROLL TO SALARY-RATE IN NEW-PAYROLL. MOVE
    EXEMPTIONS IN PAYROLL TO EXEMPTIONS IN NEW-
    PAYROLL.

ITERATION-AND-EXIT.
    WRITE NEW-PAYROLL. WRITE CHECK BEFORE ADVANC
    ING 5 LINES. READ PAYROLL-INFO-FILE RECORD; AT
    END GO TO HOUSEKEEPING-CLOSE. GO TO UPDATING-
    SSWHLD-AND-FED-TAX.

HOUSEKEEPING-CLOSE.
    CLOSE PAYROLL-INFO-FILE, NEW-PAYROLL-INFO-FILE
    WITH LOCK, PAYCHECK FILE.
```

7-7. Exercises

7-1. Write the syntactical description of a Boolean expression for COBOL which automatically preserves the precedence order of the operations.

If the value of the conditional variable **SALE** is **CHAIR**, **INVOICE** is **BED**, **SALES MAN** is **JONES**, and **CUSTOMER** is **LEE**, and if the value of the condition-name **MERCHANDISE** is **"BED"**, **BILL** is **"CHAIR"**, **BUYER** is **"LEE"**, and **SELLER** is **"JONES"**, then determine the truth value of the following Boolean expressions:

7-2. SALE IS EQUAL TO MERCHANDISE AND SALESMAN IS EQUAL TO SELLER

7-3. SALE IS EQUAL TO "CHAIR" AND CUSTOMER IS NOT EQUAL TO SELLER

7-4. INVOICE IS LESS THAN "CHAIR" OR CUSTOMER IS LESS THAN SELLER

7-5. SALE IS EQUAL TO "BED" AND SALESMAN IS EQUAL TO "JONES"

Complete the omissions in the Boolean expressions involved in the following sentences:

7-6. IF A IS EQUAL TO B THEN ADD X TO Y ELSE IF GREATER THAN C THEN ADD M TO N ELSE S1.

7-7. IF A IS EQUAL TO B THEN ADD X TO Y ELSE IF GREATER THAN ADD M TO N ELSE ADD P TO Q ELSE S2.

7-8. IF A IS GREATER THAN B OR EQUAL TO B AND X IS EQUAL TO Y THEN IF GREATER THAN B THEN ADD C TO D ELSE S1 ELSE S2.

7-9. IF A IS EQUAL TO B AND X IS GREATER THAN Y THEN ADD C TO D ELSE IF EQUAL TO THEN ADD P TO Q ELSE S1 ELSE S2.

7-10. IF A IS EQUAL TO B AND X IS GREATER THAN Y THEN ADD C TO D ELSE IF EQUAL TO THEN ADD P TO Q ELSE S1 ELSE S2.

7-11. Make flow charts for the sentences of Exercises 7-6 and 7-7.

7-12. Make flow charts for the sentences of Exercises 7-8, 7-9, and 7-10.

7-13. Write the syntactical description of an arithmetic expression for COBOL which automatically preserves the precedence order of the operators.

7-14. Explain why a period is required at the end of a conditional statement unless an else fragment follows. (HINT: Give examples of ambiguities that can result if the period is not used.)

Evaluate the following statements, where initially $A = 3.7$, $B = 6.2$, $C = 0.05$, and $D = 10$, and give the data-name of the result:

7-15. COMPUTE PRICE FROM C TIMES (B MINUS A).

7-16. COMPUTE XX6P FROM A PLUS B TIMES C DIVIDED BY D MINUS 2.

7-17. ADD A, B, D GIVING TOTAL; SUBTRACT A, B FROM TOTAL.

7-18. ADD A, B, C; SUBTRACT A, C FROM D; MULTIPLY D BY C.

7-19. MULTIPLY A BY B; DIVIDE B INTO D.

7-20. MULTIPLY A BY B; DIVIDE D INTO B.

Write statements that compute the following:

7-21. The price of 51 items at \$2.75 per dozen, where we get a 25 per cent discount.

7-22. The retail price of an item the wholesale cost of which is \$70 per dozen where it is desired to make a 20 per cent profit.

7-23. The take-home pay of an employee who works 23 hr at \$2.50 per hour and 65 hr at \$2.75 per hour if he must contribute 5 per cent of his earnings to insurance.

7-24. $\min(A^3 - BC,\ A - B/C^3)$.

7-25. $\min[\max(A,B,C),\ \max(D,E,F)]$.

7-26. Write a code to compute amortization tables.

7-27. Write a code to compute $\sqrt{N}$.

7-28. Define a new statement format that calculates a price after taking a discount.

7-29. Write a code that computes a table of retail prices as a function of per cent profit and wholesale prices.

7-30. Define a new statement format that calculates the retail price from a wholesale price and per cent profit.

7-31. Define a new statement format that gives an amortized monthly payment for a given mortgage principle, interest, and years to pay.

Write codes to evaluate the following, including the data-descriptions involving the **OCCURS** clause that may be required:

7-32. Trace **T** of a matrix **A(i,j)** where

$$\mathbf{T} = \sum_{i=1}^{n} \mathbf{A(i,i)}$$

7-33. Transpose of a matrix where

$$\mathbf{A^t(i,j) = A(j,i)}$$

7-34. The element of a matrix with the greatest absolute value.

7-35. An element of a table by the method of Fig. 3-11.

7-36. The scalar product of two vectors.

7-37. Prepare the record description that generates a table giving the prices of five different styles of women's dresses, each of which comes in six different qualities of cloth, each of which occurs in four different colors. The table is to be accessed by the style-name, cloth-name, and color-name, with a subscripted variable.

7-38. In Exercise 7-37 suppose that two of the styles were designed by **DIOR** and the other three by **CASSINI**; also suppose that three of the cloths were made by **WILSON-MILLS** and the other three by **BARKER-MILLS**; write a **REDEFINE** clause that will enable the dress price table also to be referenced by designer, cloth manufacturer, and color with a subscripted variable.

7-39. Prepare the record description for an income-tax withholding table.

7-40. Prepare the record description for a mortgage-amortization table.

7-41. Identify the parts of the example of **FD PAYROLL-DATA** in Sec. 7-5.

7-42. In the complete example of a payroll given at the end of Sec. 7-6 we neglected consideration of *new employees*, *dropped employees*, *change of number of exemptions claimed*, and *change of salary rate*. Redo the complete example to include these points. (HINT: Include an additional input data-file called **NEW-DATA**; assume that the **PAYROLL-INFO-FILE** will have the **EMPLOYEE**-records on the tape in alphabetical order by last name; assume the same is true for the employee records that will be on the **NEW-DATA** file. Be careful that the final check is written for the dropped employees.)

7-43. Redo the complete payroll example of Exercise 7-42 including an additional output tape file upon which is calculated the employer's monthly deposit of income-tax withholding, the quarterly form 941, and annual forms 940, W-3, and the annual form W-2 for each employee. (HINT: Obtain the IRS employer's instructions for reference. Note that the dropped employees must be given W-2 forms at the end of the year during which they were employed. To take care of this case, use a level 88 clause to identify active and inactive **EMPLOYEE** records. Also use the *Editing* clauses to fill out the forms properly.)

7-44. In Exercise 7-43 rewrite the payroll example including provision for hourly wages and part-time employees as well.

7-45. Include in Exercise 7-43 provision for withholding state income tax, medical insurance, and retirement.

7-46. Write a program in COBOL language for the processes illustrated in Fig. 1-12, the *inventory-control system*. (This problem will require the student first to define the ingredients involved more precisely.)

REFERENCES

The following articles and books contain discussions about the business use of computers:

Adams, Charles W.: Processing Business Data, *Control Eng.*, vol. 3, pp. 105–112, June, 1956.

ADP Experience of Local and State Governments, The, *Automatic Data Process.*, vol. 2, no. 11, pp. 25–27, November, 1960.

Angle, Carl: How TVA Built Its Data Computing Center, *Office*, November, 1958.

Anon.: Bugs in Automation, *Wall Street J.*, Dec. 27, 1960.
Boyer, Paul W.: Commonwealth Edison Starts Electronic Billing, *Elec. World*, Mar. 26, 1956.
Brown, Arthur A., and Leslie G. Peck: How Electronic Machines Handle Clerical Work, *J. Accountancy*, vol. 99, pp. 31–37, January, 1955.
Canning, R. G.: "Electronic Data Processing for Business and Industry," John Wiley & Sons, Inc., New York, 1956.
Carrol, Phil: "Cost Control through Electronic Data Processing," Society for the Advancement of Management, New York, 1958.
Churchman, C. W., and R. L. Ackoff: Operational Accounting and Operations Research, *J. Accountancy*, vol. 99, pp. 33–39, February, 1955.
Cole, John S.: Electronics for Management: Automatic Data Processing at a Class I Installation, *Armed Forces Comptroller*, vol. 3, pp. 1–8, June, 1958.
Firstman, Sidney I.: An Approximating Algorithm for an Optimum Aim-points Problem," *Naval Research Logistics Quart.*, vol. 7, no. 2, June, 1960.
Fritz, F. H.: Work Simplification through Data Processing Equipment, *Cost and Management*, September, 1958.
Gallagher, James D.: Administrative Automation at Sylvania: A Case Study—III. The Program in Operation, in "Administrative Automation through IDP & EDP," pp. 47–67, Office Management Series, no. 144, American Management Association, New York, 1956.
Geisler, M. N., and W. A. Steger: The Use of Manned Simulation in the Design of an Operational Control System, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 51–62.
Gregory, R. H.: Computers and Accounting Systems, *Accounting Research*, vol. 6, pp. 38–48, January, 1955.
———: Present and Prospective Use of Computers in Management Control, in "Computer Techniques in Chemical Engineering," pp. 21–24, American Institute of Chemical Engineers, New York, 1959.
——— and R. L. Van Horn: "Automatic Data Processing Systems," Wadsworth Publishing Co., San Francisco, 1960.
Haskins & Sells: "Introduction to Data Processing," Haskins & Sells, New York, 1957.
Hattery, L. H., and G. P. Bush: "Electronics in Management," University Press, Washington, D.C., 1956.
Hetrick, James C.: Mathematical Models in Capital Budgeting Can Be of Great Aid in Managerial Decisions, *Harvard Business Rev.*, vol. 39, no. 1, pp. 49–64, January–February, 1961.
Hildebrandt, Paul, and H. Isbitz: Radix Exchange—An Internal Sorting Method for Computing Machinery, *J. Assoc. Computing Mach.*, vol. 6, no. 2, pp. 156–163, April, 1959.
Hindle, R.: A Bank Adopts Automatic Data Processing, *Computer J.*, vol. 3, no. 3, pp. 127–130, October, 1960.
Hopper, Grace Murray: Automatic Coding for Digital Computers, *Computers and Automation*, September, 1955.
———: Automatic Programming for Business Applications, *Proc. Fourth Ann. Computer Appl. Symposium*, 1958, Chicago, pp. 45–50.
Jones, W. T., Jr.: Inventory Control at Knitting Mills, *Data Processing*, vol. 2, no. 7, pp. 31–32, August, 1960.
Karp, Harry R.: Using Corporate Models for Business Control, *Control Eng.*, vol. 7, pp. 135–137, September, 1960.
Keenoy, C. L.: The Impact of Automation on the Field of Accounting, *Accounting Rev.*, April, 1958.

Kibbee, J. M.: Management Games and Computer, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 11–16.
Kozmetsky, G., and Paul Kircher: "Electronic Computers and Management Control," McGraw-Hill Book Company, Inc., New York, 1956.
Kramer, S. J.: Processing an Average of 90,000 Checks and Deposit Transactions Daily, *Computers and Automation*, January, 1960, pp. 10–12.
Kruse, Benedict: Electronic Brain Keeps Tabs on 11,500 Rexall Stores, *Am. Business*, vol. 24, pp. 41–47, December, 1954.
Laden, H. A.: Automatic Programming, Fact or Fancy? *Management and Business Automation*, vol. 1, pp. 29–35, February, 1959.
Lassing, Lawrence P.: Computers in Business, *Sci. Am.*, vol. 190, pp. 21–25, January, 1954.
Laubach, Peter B.: "Company Investigations of Automatic Data Processing," Graduate School of Business Administration, Harvard University, Boston, 1957.
Leveson, J. H.: "Electronic Business Machines," Philosophical Library, Inc., New York, 1959.
Levin, Howard S.: "Office Work and Automation," John Wiley & Sons, Inc., New York, 1956.
Lewis, Daniel C.: An Iron Foundry's Computer Applications, *Office*, March, 1958.
McCracken, D. D., H. Weiss, and Tsai-Hwa Lee: "Programming Business Computers," John Wiley & Sons, Inc., New York, 1959.
Minor, Robert: MATS—A Global Operation Backed by Data Processing, *Management and Business Automation*, vol. 4, pp. 18–23, December, 1960.
Muns, Frank H.: Using IDP for Inventory Control, *Controller*, February, 1959.
Oliver, Foster F.: Preparing for Random Access Equipment for Ordnance Inventory Control, *NAA Bull.*, January, 1959.
Pardee, D. R.: Real-time Management Control at Hughes Aircraft, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 603–608.
Price, Waterhouse & Co.: "The Auditor Encounters Electronic Data Processing," International Business Machines Corp., New York (not dated).
Report to Conference on Data Systems Languages, April, 1960, Government Printing Office #1960 0-552133.
Rotkin, I.: The Mechanization of Letter Mail Sorting, *Proc. Eastern Joint Computer Conf.: Computers with Deadlines to Meet*, 1958, New York, pp. 54–57.
Rowe, A. J.: Computer-based Management Control, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 582–592.
Salveson, M. E.: Electronic Computers in Business, *J. Ind. Eng.*, March–April, 1958.
Scott, A. E.: Automation Preparation of Flow Chart Listings, *J. Assoc. Computing Mach.*, vol. 5, no. 1, pp. 57–66, January, 1958.
Semment, J. E.: A Method of Combining ALGOL and COBOL, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 379–389.
Steel, T. B.: UNCOL—Universal Computer Oriented Language Revisited, *Datamation*, vol. 6, no. 1, pp. 18–20, January–February, 1960.
Stoller, David S., and R. L. Van Horn: Design of a Management Information System, in C. West Churchman and Roger Craine (eds.), "Management Technology," Institute of Management Science, White Plains, N.Y., 1959.
Swanson, P. R.: Information Retrieval, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 239–248.
Taylor, J. C.: Data Collection as a By-product of Normal Business Machine Operation, *Proc. Western Joint Computer Conf.*, 1955, pp. 34–41.
Toan, A. B., Jr.: Auditing, Control and Electronics, *J. Accountancy*, vol. 99, pp. 40–45, May, 1955.

Use of Electronic Data-processing Equipment (hearing before Sub-committee on Census & Government Statistics of the Committee on Post Office and Civil Service, House of Representatives), June 5, 1959.

Vazsonyi, A.: An On-line Management System Using English Language, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 17–38.

Wiseman, R. T.: Life Insurance Premium Billing and Combined Operations by Electronic Equipment, *J. Assoc. Computing Mach.*, vol. 1, pp. 7–12, January, 1954.

The following periodicals frequently contain articles or reviews of articles on the application of computers to business-oriented problems:

Accounting Review, quarterly, American Accounting Association.
Banking, weekly, McGraw-Hill Publishing Company, Inc., New York.
Chemical Processing, monthly, Putnam Publishing Company.
Computer Journal, quarterly, British Computer Society, Ltd., London.
Computing Reviews, bimonthly, Association for Computing Machinery, New York.
Cost Accountant, monthly, Institute of Cost and Works Accountants.
Cost and Management, monthly, Society of Industrial & Cost Accountants of Canada.
Datamation (*Research and Engineering, The Magazine of Datamation*), bimonthly, Relyea Publishing Co., New York.
Factor Management and Maintenance, monthly, McGraw-Hill Publishing Company, Inc., New York.
Harvard Business Review, monthly, Soldiers Field Road, Boston.
Journal of Accountancy, monthly, American Institute of Accountants.
Management Review, monthly, American Management Association.
New York Certified Public Account, monthly, New York State Society of Certified Public Accounts.

CHAPTER 8

PROGRAMMING TO ACHIEVE INTELLIGENCE

8-1. Advanced Concepts in Automatic Programming

Achieving Intelligence. The purpose of this chapter is to introduce some of the advanced concepts in automatic programming, with particular emphasis on those aspects that deal with programming to achieve artificial intelligence. When we discuss programming methods intended to impart a measure of "intelligence" to a computer, we do not mean the simulation of actual mental processes involved, but rather their rational reconstruction in an attempt to mimic their outputs or their structures. The goal is to determine effective programming techniques for accomplishing "humanoid"† tasks, imitating the human mental processes only when this proves to be the most effective way to do the job.

Syntactical techniques prove to be of great value in programming to achieve intelligence. Therefore we devote the next two sections to a discussion of syntactical analysis, in particular as applied to automatic-programming-language translation. Besides presenting a more detailed study of the syntactical structures themselves, this discussion is of great importance in its own right for its consideration of automatic-programming-translation methods. In Secs. 8-2 and 8-3 we shall see how a single translator program can be utilized to translate any automatic program into a machine language, provided the automatic-programming language is suitably syntactically defined.

Next, the discussion turns to the problems of deductive and inductive inference. Deductive inference is the process of drawing consequences or conclusions from given premises, while inductive inference is the process of attempting to determine hypotheses or theories from which the given propositions may be deduced (see Sec. 10-13). Methods of inductive inference—namely, the so-called "heuristic programming"—have received recent prominence as being associated with models of intelligent behavior. However, these developments should not overshadow the more "conventional" mathematical methods in the area of artificial intelligence. These "conventional" methods, such as logic, probability, linear and dynamic

† We borrow this term from H. A. Simon. See his Modeling Human Mental Processes, *Proc. Western Joint Computer Conf.*, May, 1961, pp. 111–119.

programming, game theory, etc., are associated with deductive inference and are the topic of Secs. 8-4 and 8-5. In Sec. 8-5 we apply such mathematical methods of deductive inference to formulating the humanoid task of making a medical diagnosis, interpreting in quantitative terms many of the decisions the physician now makes in a qualitative form.

Having emphasized the more conservative mathematical approach, we can now turn, with a clear conscience, to the problems of inductive inference and the newer techniques of heuristic programming. The concept is introduced in Sec. 8-6; and the so-called "general problem solver" is illustrated in Sec. 8-7. Heuristic programming was developed in order to study the manner in which human problem solving is accomplished, such as the playing of games or the proving of geometric theorems or trigonometric identities. This field holds great promise for forming the basis of techniques that may enable truly intelligent problem solving by computers.

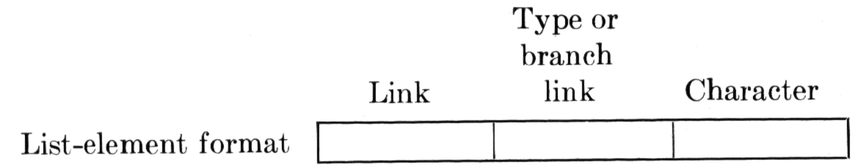
The final section of this chapter is devoted to the consideration of abstraction and creativity. The problem of programming computers to conceive an abstraction will be approached through the use of syntactical analysis coupled with the methods of deductive and inductive inference. There are two closely related aspects to the problem—that of *recognizing* an object as a concrete example of an abstraction, and that of *learning*, or formulating abstract concepts from the characteristics of a given population. The problem has not yet been solved, of course, but we hope to create an awareness of its existence and to present at least a glimmer of possible promising methods of approach.

The most human of humanoid tasks is perhaps that of creativity; hence we end the chapter (Sec. 8-8) with a short discussion of an example of creating a musical composition by a computer and a discussion of the problems that confront the programmer if true creativity through computer programming is to be achieved. In presenting this material, the author must act as a reporter describing current research approaches, attempting to stimulate the reader to delve further into these new and untried areas on the research frontiers.

List Processing. Programming to achieve intelligence characteristically involves extensive symbol manipulation, and a technique is required for handling structures that evolve during the course of processing. For these purposes, automatic list-processing programs have been developed. The threaded-list and push-down-list concepts introduced in Sec. 5-5 are the basic ingredients of such an automatic-programming language. It is appropriate to make some further remarks here about list processing, since it will be understood throughout the rest of this chapter that *list-type structures will be handled through list processing wherever required.*

As a brief illustration of how a list-processing type of language works,

consider the list element of Fig. 5-8 (page 178) and call the three components of an element the *link*, the *type or branch link*, and the *character*, this latter being *B*, *C*, or *E*, as follows:



We may let the entire memory be composed of lists. The *instruction* list, having elements (i.e., instructions) of a different format from the rest of the data, will be processed separately. The unused memory words will comprise a push-down list. When the address of a list (i.e., the address of the first element of a list or sublist) starts with *P*, it is automatically understood to be a push-down list.

Although we cannot here develop a full list-processing language, we shall give examples of the kinds of instructions required in such a language, illustrated by the manipulation of the lists of cards associated with the well-known game of rummy. The original deck of cards, assumed to be already shuffled, forms a push-down list, called the *stock*, to which we give the address *P*STK. Suppose there are four players and each is dealt seven cards in a hand; let the addresses of the hand lists be HA1, HA2, HA3, and HA4. Then, in order to deal, we would need an instruction of the type

Move element *P*STK to end of list HA*I*

Of course, the next element of *P*STK would pop up. Initially $I = 1$, but I would be increased by 1 for each repeat of this instruction (i.e., the cards would be dealt one at a time to each successive player) until $I = 4$, when I would be reset to 1 again. When 28 cards have been dealt, the 29th card would be turned up to start the *discard* list, of address *P*DSC, by an instruction

Move element *P*STK to front of list *P*DSC

Note that an element within a push-down list would never be directly addressed, by the very nature of the push-down-list concept. Now we are ready for the first play. The first player decides whether to pick a card from the discard list or the stock list. If, for instance, he desires to choose the card from the discard list, an instruction of this type would be used:

Move element *P*DSC to front of list HA*I*

Then the player examines his hand to see whether or not he has three or four of a kind, or a run of three or more in the same suit. This would be

accomplished by instructions of the kind

Compare the type of the ith element of list HAI with the type of the jth element of list HAI

If three or four of a kind, or a run of three or more, are found, then a new list is initiated, called a *spread* list, SPJ, with an instruction of the form

Move elements i through j of list HAI to front of list SPJ

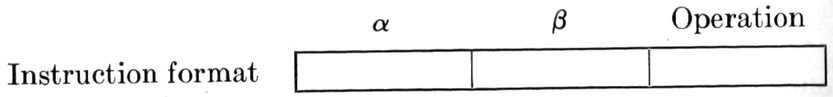
where J will be given a different value of each spread. The second player would go through the same routine, etc. When a player has no cards remaining in his hand he stops the game and counts his winning points. Hence we would need an instruction of the type

Find the number of elements in the list HAI

When looking for three or four of a kind or for runs of three or more, a reshuffling of a hand may be convenient. This would require instructions of the type

In list HAI find the address of element in list with type T
In list HAI remove element i and insert it after element j

Note that, if a symbolic machine-language instruction type were desired, it would be natural in our case to have a two-address instruction format of the form



However, some of the instructions described above have up to four addresses. This could be taken care of by having several index registers to store the extra required addresses temporarily. Of course an address-assigner routine of the type described in Sec. 5-7, in combination with a simulational routine of the type described in Sec. 5-4, would be required to compile or interpret such two-address symbolic machine-language instructions.

8-2. Automatic-programming-language Translation

Syntax-directed Automatic-programming-language Translation. In Chap. 5 we briefly described some methods that could be used to compile automatic-language "pseudoinstructions" in terms of machine-language instructions. For each pseudoinstruction of the automatic language, a subroutine is written that can interpret, or "translate," the pseudoinstruction into a sequence of machine-language instructions. In this section, on the other hand, we shall describe an entirely different concept for automatic-programming-language translation,† which requires only a single subroutine to perform the entire translation. The different pseudoinstructions are characterized by their respective syntactical and semantic descriptions, and the single subroutine is designed to translate *any* such syntactical and semantic description into the machine-language instructions.

The advantage of such a concept is that changes and modifications in the automatic-programming language can be made without regard to the machine-language-translation problem, for such changes would involve the alteration of only the syntactical and semantic description of the language. In fact, the same translator program would be able to translate many different automatic-programming languages (that are suitably defined) into machine language. Also, the translator program can translate from one automatic-programming language to another automatic-programming language. This latter consideration would be of importance when, for example, it is desired to translate a general language such as ALGOL or COBOL into a specific automatic language, which may have been devised by the computer manufacturer, etc. Hence the method offers a possible solution to the burgeoning "tower-of-Babel" phenomenon that is occurring in the automatic-programming-language field (see Fig. 8-1).

To describe this new approach, it will be convenient to define a few new terms for old ideas. Language translation is said to take a *source language* into a *target language.*‡ In our case, the source language is the original automatic-programming language, and the target language is the machine language (or other automatic-programming language) in terms of which the final translated result is to appear. By the *definition of a language* we mean the syntactical and semantic description of the language. Note that the definition of a language must be stated in terms of some other language. For example, our definitions of ALGOL and COBOL of the previous chapters were stated in terms of English. However, for language-translation purposes, *the definition of the source language must be given in terms of the target language.* In fact, much of the burden of the success of the translation process is placed on the accuracy and completeness with which these automatic-language syntactical and semantic descriptions are made. The *translator* is the program, written in a metalanguage, which operates only on syntactical and semantic structures.

† The method described is based on a concept of E. T. Irons; see E. T. Irons, A Syntax Directed Compiler for ALGOL 60, *Communs. ACM*, vol. 4, no. 1, pp. 51–58, January, 1961.

‡ If the target language is the final machine-language code, the term *object language* is frequently used.

Symbolically, we can represent the situation as follows: Let $\mathcal{S}(\mathcal{E})$ and $\mathcal{I}(\mathcal{E})$ be the definitions of the source language and target language, respectively, in terms of English, such as we have given in previous chapters for ALGOL and COBOL. For our translation method, we really desire the definition $\mathcal{S}(\mathcal{I})$—that is, the definition of the source

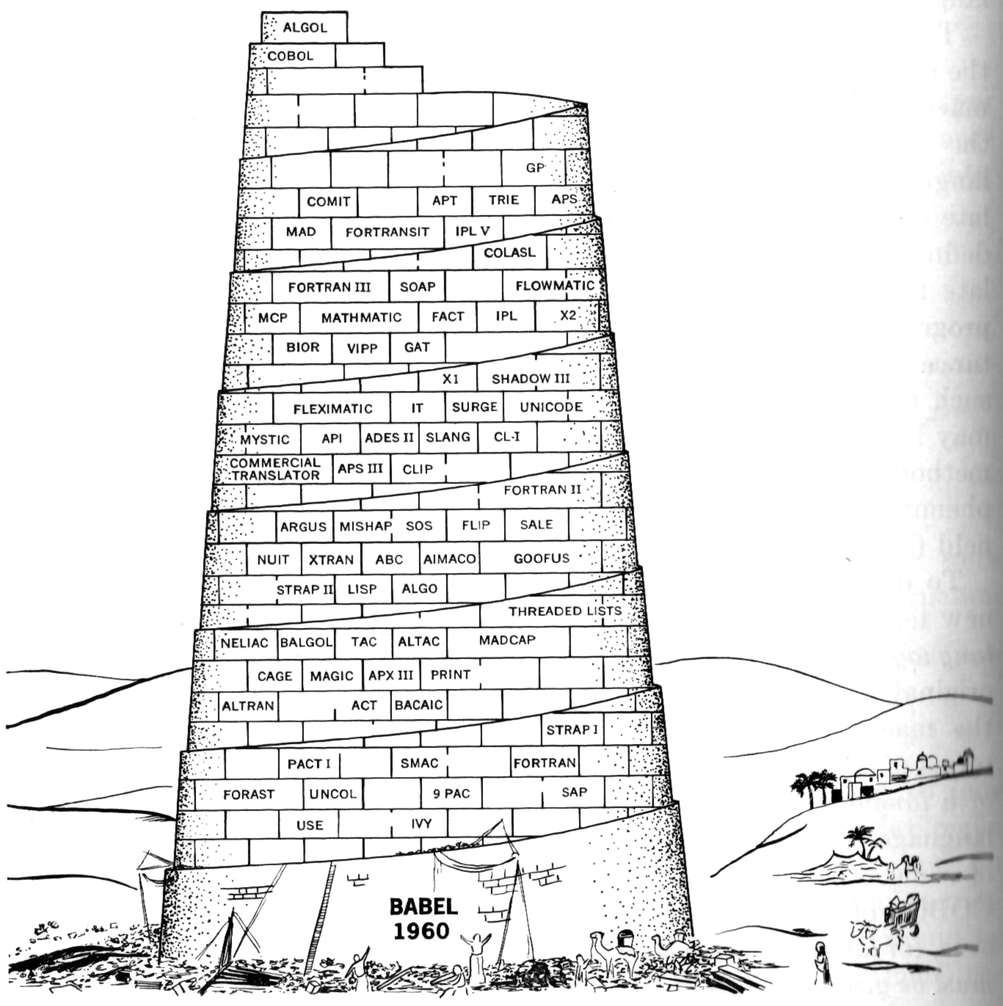


FIG. 8-1. "Tower of Babel," illustrating the programming-language-translation problem. The expressions in the blocks of the tower are names of just a few of the existing automatic-programming languages. (*Adapted from the January*, 1961, *cover of Communs. ACM, vol.* 4, *no.* 1, *with permission.*)

language in terms of the target language—where $\mathcal{I} = \mathcal{I}(\mathcal{E})$ is given by, say, the computer manufacturer's programming manual. Now the translator can be considered as a transformation T such that $T[\mathcal{S}(\mathcal{I})] = \mathcal{I}$, with the property that, if $\mathcal{P}(\mathcal{S})$ is a program written in the source language, then $T[\mathcal{P}(\mathcal{S})] = \mathcal{P}[T(\mathcal{S})] = \mathcal{P}(\mathcal{I})$ [where $\mathcal{P}(\mathcal{I})$ is the translated program in terms of the target language]. Thus *the translator operates by "looking at"*

the successive symbols of a program written in the source language, comparing these symbols with those of the definition of the source language, and transforming them into symbols of the target language, according to the syntax and semantics of the description (see Fig. 8-2).

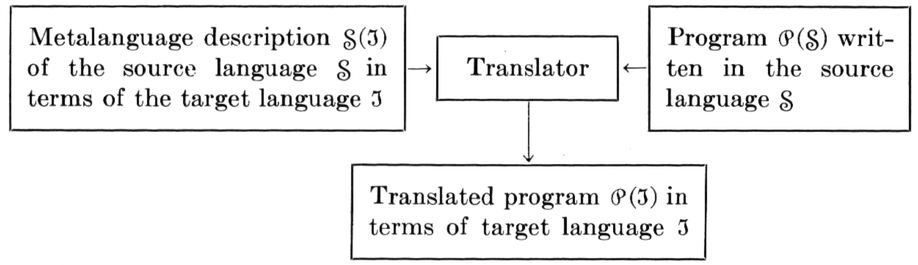


FIG. 8-2. Use of translator: $T\{\mathcal{P}[\mathcal{S}(\mathcal{I})]\} = \mathcal{P}\{T[\mathcal{S}(\mathcal{I})]\} = \mathcal{P}(\mathcal{I})$.

The Syntactical and Semantic Descriptions. We shall introduce the notion of the syntactical and semantic descriptions by means of the

TABLE 8-1. SYNTAX AND SEMANTICS FOR TRANSLATING ENGLISH INTO KOREAN†

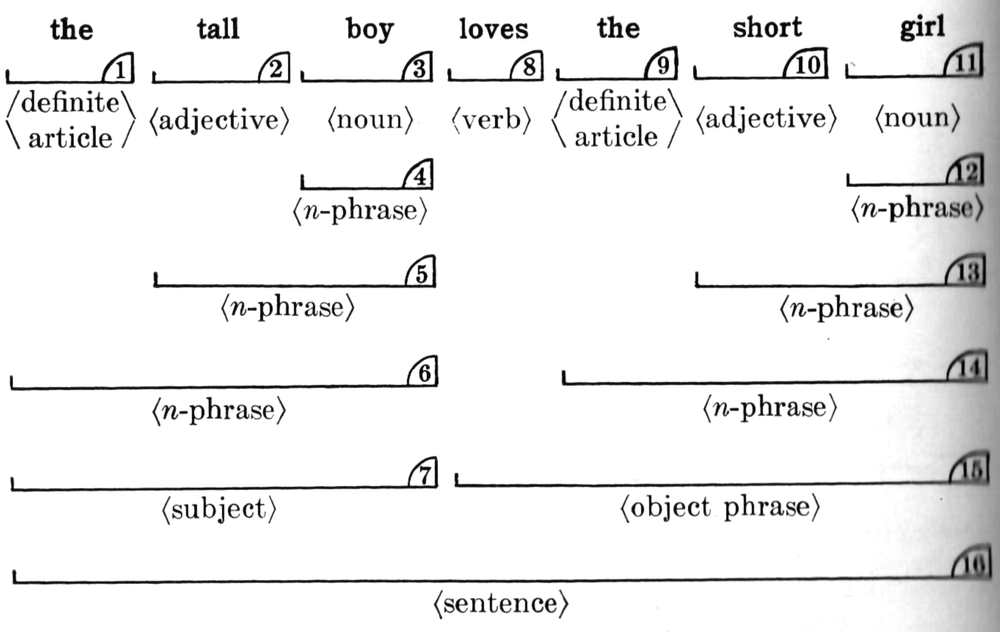
	Syntax	*Semantics*
1	⟨noun⟩ : : = **boy**|**girl**	⇒ 나 남자 | 여 자
2	⟨adjective⟩ : : = **tall**|**short**	⇒ 높 | 짧 을
3	⟨definite article⟩ : : = **the**	⇒ ⟨empty⟩
4	⟨verb⟩ : : = **loves**	⇒ 사 랑 하 다
5	⟨*n*-phrase⟩ : : = ⟨noun⟩ | ⟨definite article⟩⟨*n*-phrase⟩ | ⟨adjective⟩⟨*n*-phrase⟩	⇒ {C_1}|{C_1} | {C_2} 은 {C_1}
6	⟨object phrase⟩ : : = ⟨verb⟩⟨*n*-phrase⟩	⇒ {C_1} 를 {C_2}
7	⟨subject⟩ : : = ⟨*n*-phrase⟩	⇒ {C_1} 가
8	⟨sentence⟩ : : = ⟨subject⟩⟨object phrase⟩	⇒ {C_2} {C_1}

† Grossly simplified for illustrative purposes.

example of translating English as the source language into Korean as the target language. It is important to observe, however, that, even though we are using a written-language translation as an example, it can be shown that the method to be presented is not sufficient for actual human-language translation, although clearly the method is one of the ingredients of that process. Indeed, the illustration is intended only to emphasize the important aspects of automatic-programming-language translation. The syntactical definitions given in Table 8-1 indicate the grammatical parts required to translate the sentence

the tall boy loves the short girl

The metalanguage notation is the same as that used in the descriptions of ALGOL and COBOL: e.g., the first line says that a "noun" can be either **boy** or **girl**; the fifth line states that an "*n*-phrase" (noun phrase) can be an adjective followed by an *n*-phrase, or a definite article followed by an *n*-phrase, or a noun. Thus we could analyze (or diagram) the sentence as follows, according to the recursive use of these rules of syntax:



Although we have used this process in Chap. 6, let us observe the successive steps more closely (see numbers on the brackets): (1) Choosing the first word of the sentence, namely, **the**, we look on the *right-hand side* of the *syntax* of Table 8-1 to identify its syntactical form; the lines of Table 8-1 are observed successively *from the top down;* line 3 indicates that **the** is a "definite article." The next step is to determine whether or not a "definite article" can be developed as a part of a higher form,

starting again from line 3 of the syntax and working down. Line 5 has the term as part of the second alternative from the right—if an *n*-phrase follows **the**, then we can incorporate it into a larger *n*-phrase. (2) To see whether or not this development can be made, we choose the next word of our sentence, namely, **tall**, and establish it as an adjective; the first (rightmost) alternative of line 5 could be used to incorporate **tall** into a larger form if we only had an *n*-phrase to its right. Hence (3) we go on to the next word of the sentence, namely, **boy**. By syntax line 1, this is a noun; by the third alternative from the right on line 5, we find (4) that the noun can be called an *n*-phrase; again by syntax line 5, first alternative on the right, we can (5) combine our adjective (**tall**) with the following *n*-phrase (**boy**) to form a larger *n*-phrase, and so forth. Note that we could not use line 7 yet, because we must use the syntax rules in precedence order from top to bottom, starting each time at the line at which the previous syntax form was completed. Hence we can use line 7 only when we have exhausted all possibilities of previous occurrences of an *n*-phrase in a syntactical rule. This occurs after we have made **the tall boy** an *n*-phrase, because this *n*-phrase is not preceded by another adjective or a definite article (as required by line 5) or by a verb (as required by line 6).

Each syntax line of Table 8-1 has an associated semantic part, to the right of the ⇒, giving the Korean description of the corresponding syntax alternatives. For instance, the semantics of line 1 of Table 8-1 gives the Korean words for **boy** and for **girl**; line 3 states that Korean has no definite article. The semantic descriptions of lines 5 through 8 require further explanation: the $\{C_n\}$ stands for the Korean translation (or target-language *code*) of the corresponding grammatical form—i.e., of the word that the corresponding grammatical form represents at the time of interrogation. For example, consider the noun **boy**; in using line 5 to form an *n*-phrase of this noun, the corresponding $\{C_n\}$ represents the Korean translation of **boy** (see line 1). The reason for the subscript is that frequently more than one term or component enters into a syntax alternative, e.g., ⟨adjective⟩⟨*n*-phrase⟩ in line 5 of Table 8-1. To distinguish between the translations of each of these components, we number each component of an alternative from *right to left* and index the $\{C_n\}$ accordingly. For example, consider the first (rightmost) alternative of line 5, together with its corresponding semantic description. Korean is written in columns (like Chinese) rather than in rows (as English is). The semantics says that an adjective followed by an *n*-phrase is translated into Korean with (the Korean translation of) the adjective on top, the symbol 은 under this, and then (the Korean translation of) the *n*-phrase under that. In Fig. 8-3 we have completed the translation of our sentence according to the syntactical analysis described above, in conjunction with the semantic descriptions given in Table 8-1. For a thorough under-

standing of this relatively simple process, the student should follow *in detail* each of the steps of Fig. 8-3 as indicated by the numbers on the brackets.

Note that in Table 8-1 we have numbered the lines, or "levels," ascribing "level numbers" to the terms on the left of the syntax descriptions. These level numbers can entirely replace the terms in such definitions, as shown in Table 8-2. The boldface words can be considered as "literals,"

TABLE 8-2. LEVEL-NUMBER REPRESENTATION OF THE SYNTAX OF TABLE 8-1

1 : : = boy|girl
2 : : = tall|short
3 : : = the
4 : : = loves
5 : : = 1|3,5|2,5
6 : : = 4,5
7 : : = 5
8 : : = 7,6

since they are not generic entities but are actual words of the source language. The use of level numbers assists the analysis, provided we assign the level number "**0**" to the literals. The analysis progresses in a manner entirely equivalent to that described above, where now we think of each symbol or combination of symbols in the sentence as being successively *replaced* by one with a *higher* level number in accordance with the syntax rules, in an attempt to obtain the highest level number which will replace the entire sentence. Of course, each time a replacement is made the corresponding semantic descriptions, or translations, are aligned, as illustrated for our example in Fig. 8-3. For this example, the successive transformations of the analysis would appear as follows:

Step number	**the tall boy loves the short girl**
1	**3** tall boy loves the short girl
2	**3 2** boy loves the short girl
3	**3 2 1** loves the short girl
4	**3 2 5** loves the short girl
5	**3 5** loves the short girl
6	**5** loves the short girl
7	**7** loves the short girl
8	**7 4** the short girl
9	**7 4 3** short girl
10	**7 4 3 2** girl
11	**7 4 3 2 1**
12	**7 4 3 2 5**
13	**7 4 3 5**
14	**7 4 5**
15	**7 6**
16	**8**

The student should trace each step in conjunction with Table 8-2.

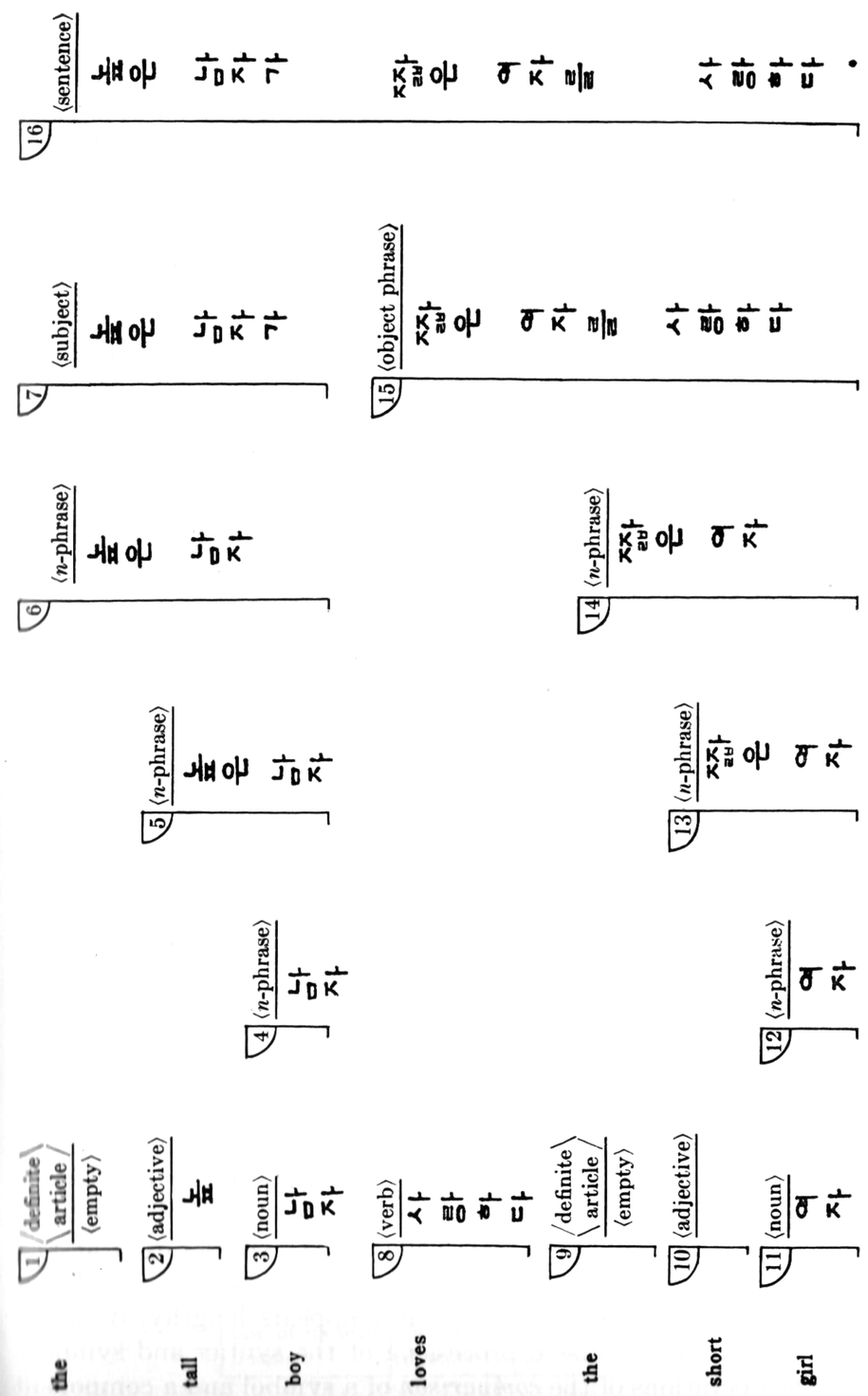


FIG. 8-3. Successive stages in the translation of an English sentence into Korean, for our illustration. Above the horizontal lines are shown the successive changes in the syntax source-language list and below each is shown the Korean translation. The numbers on the brackets indicate the order in which the steps are made.

The Translator. With the experience gained in our English-to-Korean example we can now proceed to describe the translator itself in detail. The translator simply manipulates a list and its sublists. The list is the *source* (-language) *code list*, which initially is the sentence (or code) in the source language (English); the sublists, called the *target* (-language) *fragments*, are the translated fragments in the target language (Korean) which are generated as the syntax analysis takes place. Each time a larger syntax entity (i.e., one of higher level number) is found, *the larger entity replaces the components of which it is made* in the source code list and the associated translated parts combine to form larger target fragments. Finally, the source code list becomes a single element—namely, the "sentence"—and the associated translated target fragments merge to form the list that is the desired translation. Of course, the level numbers themselves can be used in the source code list instead of their corresponding syntactical entities, as illustrated above.

Before proceeding further, let us more clearly define terms that describe the parts of a syntax definition. Each definition is a *line*, and the line number is taken as the *level number* of the corresponding syntactical entity (on the left of the : : =) of that line. The possible forms listed on the right of the syntax definition (to the right of the : : =) are called *alternatives;* the entities that make up an alternative are called *components.* Thus in the line

$$\langle n\text{-phrase}\rangle \,::=\, \langle \text{noun}\rangle \,\Big|\, \begin{matrix}/\text{definite}\backslash\\ \backslash\text{article}/\end{matrix} \langle n\text{-phrase}\rangle \,|\, \langle \text{adjective}\rangle\langle n\text{-phrase}\rangle$$

there are three alternatives, namely,

$$\langle \text{noun}\rangle \qquad \begin{matrix}/\text{definite}\backslash\\ \backslash\text{article}/\end{matrix} \langle n\text{-phrase}\rangle \qquad \text{and} \qquad \langle \text{adjective}\rangle\langle n\text{-phrase}\rangle$$

and the two components of the rightmost alternative are

$$\langle \text{adjective}\rangle \qquad \text{and} \qquad \langle n\text{-phrase}\rangle$$

The components can be expressed directly as their respective level numbers. The source code being translated is said to be composed of a list of *symbols* (words), where a symbol can be a *literal* or a *level number.* The level number of a literal is assumed to be zero. We shall denote by *key symbol* that symbol of the source code under consideration at any stage of the analysis process.

Although the description of the translator appears lengthy, it carries out a relatively simple recursive processing of the syntax and symbols, using only the operations of the *comparison* of a symbol and a component and the *selection* of an adjacent symbol or component for processing. Figure 8-4 gives the rough flow chart of the translator. Box 1 gives the

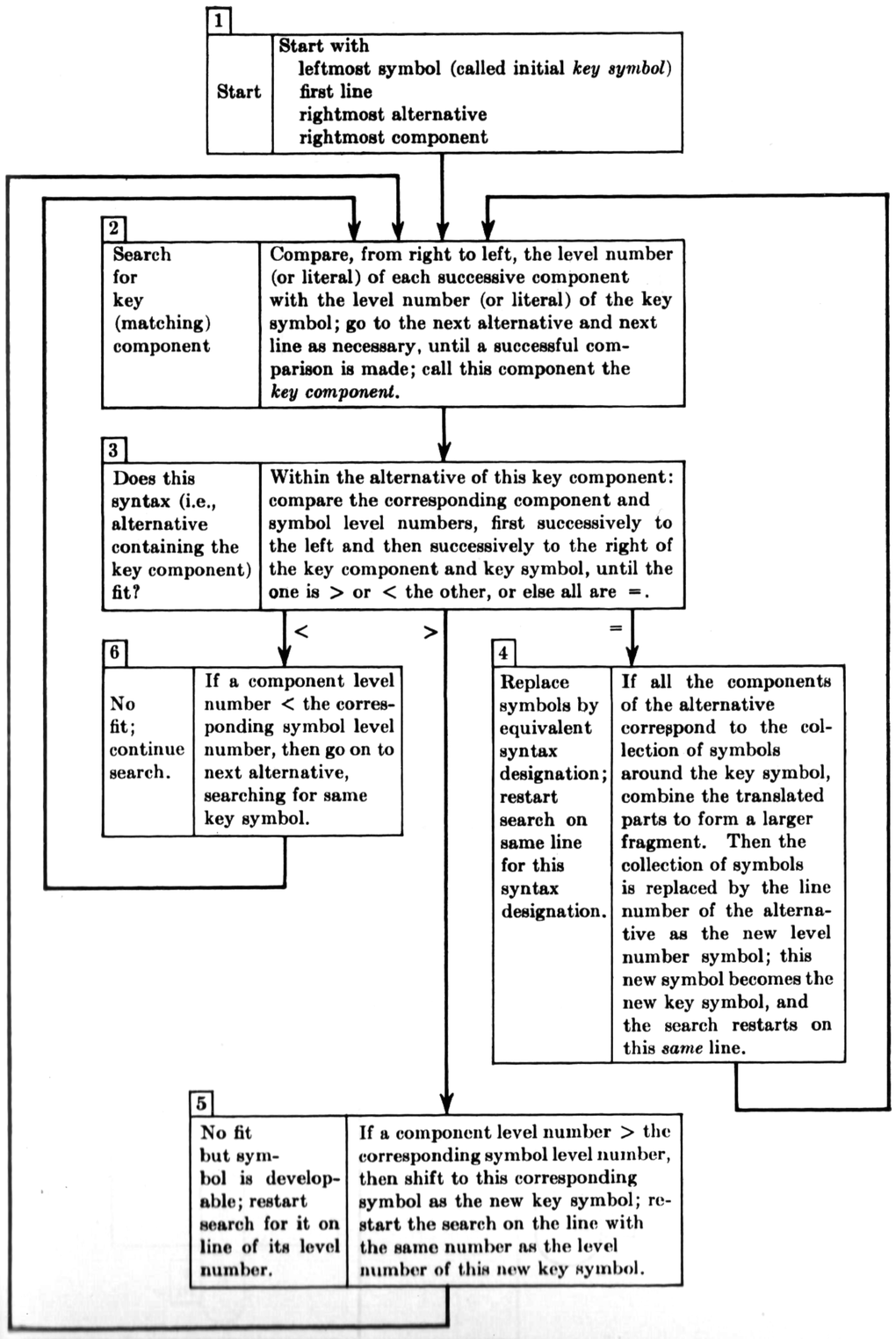
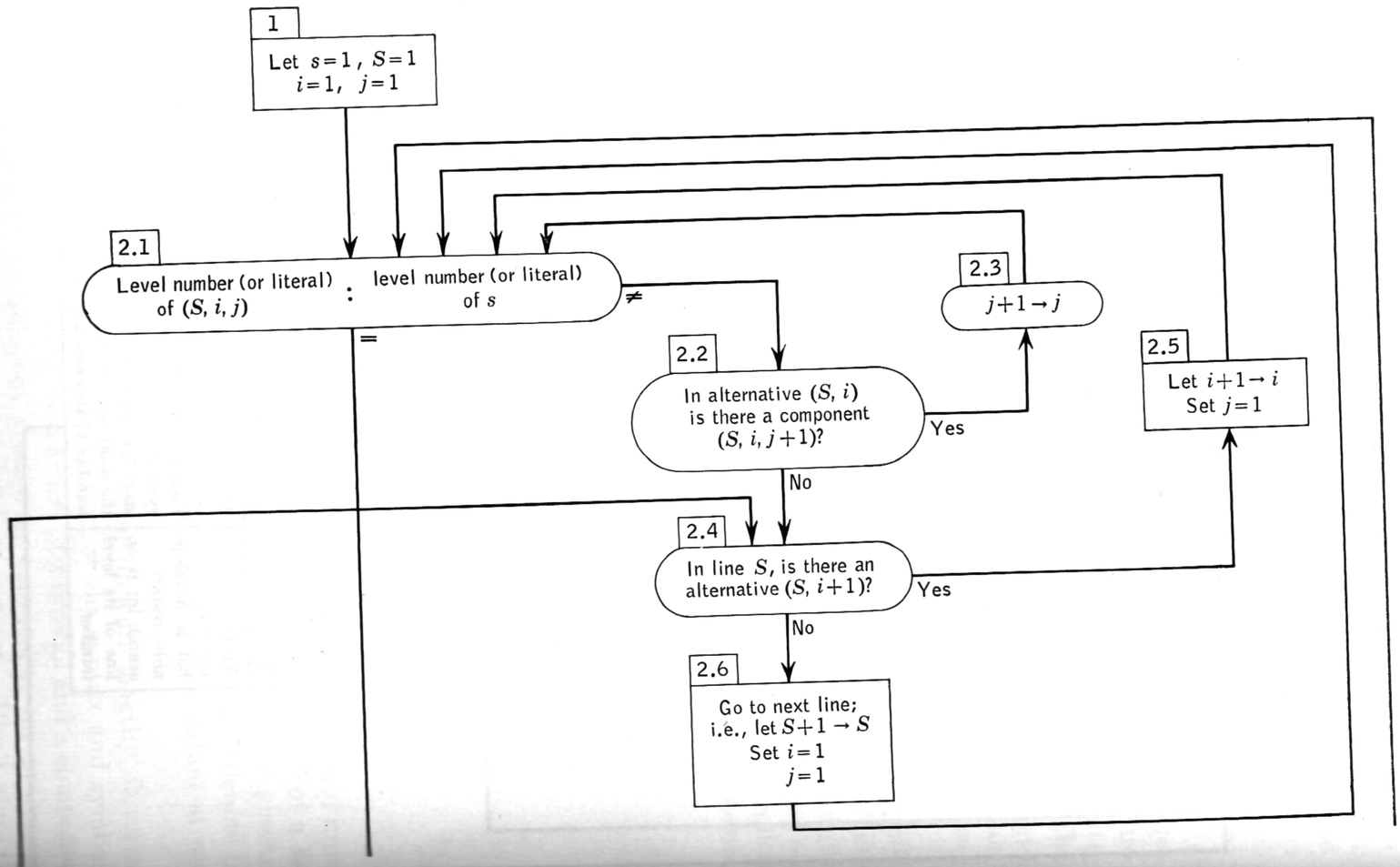


FIG. 8-4. Gross flow chart of the translator.



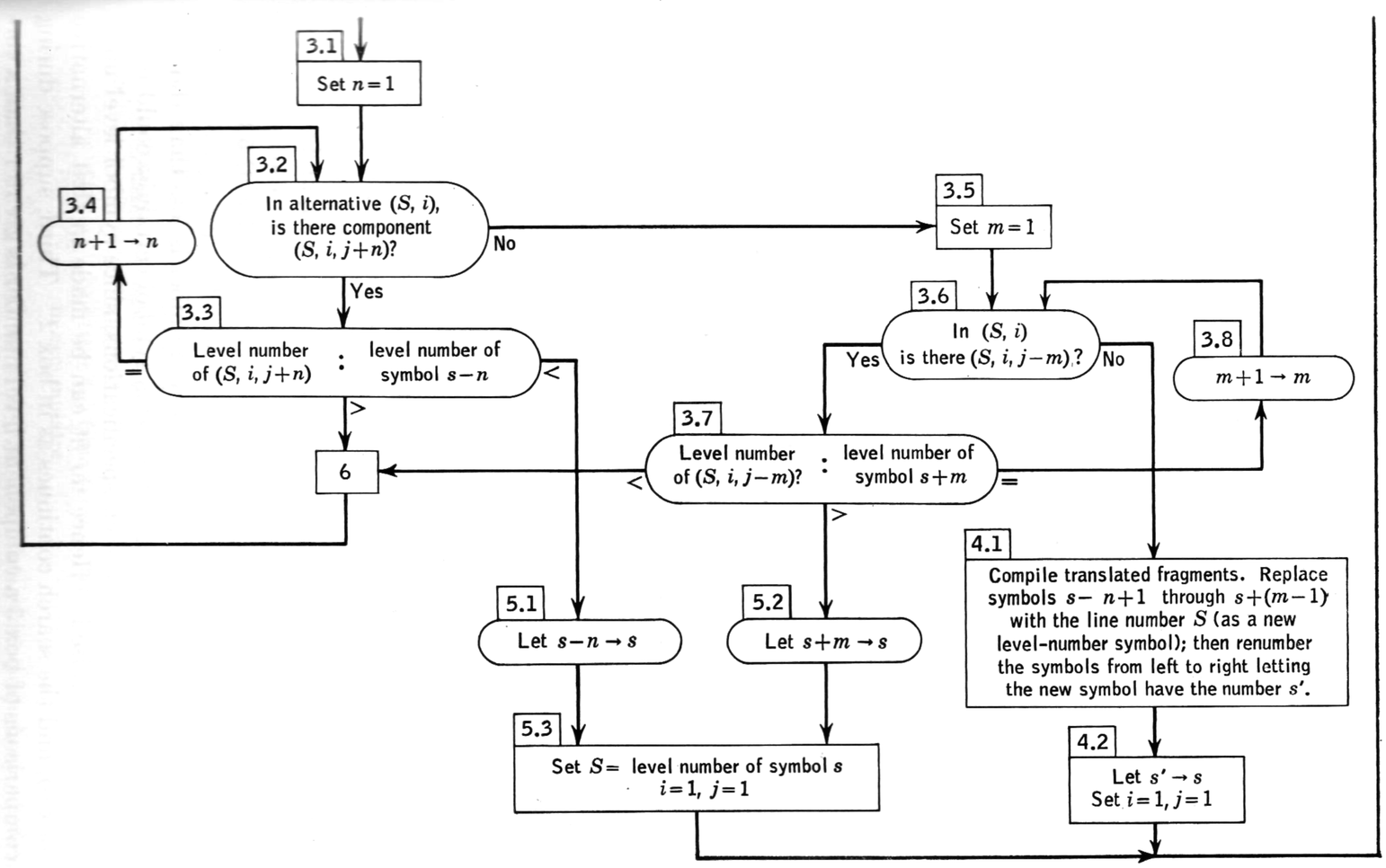


FIG. 8-5. Full flow chart of translator, utilizing index notation.

starting point: the initial key symbol is the leftmost symbol of the original source code to be translated, and the first component under consideration is the rightmost component of the rightmost alternative of the first syntax line. Box 2 describes the search through the successive components for one that will match the key symbol, the search proceeding from right to left from the top line down. (The component with which the match is made is then called the *key component.*)

Recall from our above examples that a syntactical replacement can be made only when all the components of an alternative completely match, or "fit," a set of adjacent symbols of the source code. Box 3 systematically performs the necessary comparisons, first to the left and then to the right of the key symbol and of the key components. In our above example, comparisons around the key component never occurred to the left and right at the same time; but in general both directions can be required when there are three or more components in an alternative (see the examples in the next section). For instance, suppose that the key component ⟨kcs⟩ were in the alternative

$$\cdots \mid\langle bc\rangle\langle ac\rangle\langle kcs\rangle\langle xc\rangle\langle yc\rangle\mid \cdots$$

and that the key symbol (which must be equal to the key component) were in the source code list

$$\cdots \langle bs\rangle\langle as\rangle\langle kcs\rangle\langle xs\rangle\langle ys\rangle \cdots$$

Then the comparisons of components and symbols would be made in the following order: ⟨ac⟩ with ⟨as⟩; ⟨bc⟩ with ⟨bs⟩; ⟨xc⟩ with ⟨xs⟩; and ⟨yc⟩ with ⟨ys⟩.

Actually, only the level numbers need be compared, and three cases may result. First, suppose every component (level number) in the alternative of the key component is equal to its corresponding symbol (level number) in the set about the key symbol in the source code; then a higher level number, namely, the number of the line of the alternative, replaces the set of corresponding source code symbols. The previous translation fragments associated with all these symbols are combined, and the search of box 2 is restarted at this same line (as in our example above, where **2** and **5** were replaced by **5**, and we used line 5 to replace **3** and **5** by **5**, and so forth); this is accomplished in box 4. Second, suppose that during the comparisons of box 3 a component level number is less than the corresponding symbol level number; then this symbol certainly could never be developed so as to meet such specifications, since symbol level numbers can only be raised. Hence no fit can be made in that alternative (see box 6), and the search continues as in box 2. Third, suppose during the comparisons of box 3 a component level number is greater than a corresponding symbol level number; this indicates that perhaps the symbol level number could be developed to match the higher component level number. Hence the procedure of box 5 is followed, and this symbol is chosen as the new key symbol; the search of box 2 is restarted on the line having the same level number as this new key symbol.

If the syntax is properly formulated and the sentence, or source code, is properly formed, then the process should end with the entire source code having been replaced by the highest level number and the translation—which is going on with each replacement—having been completed. The student is now in a position to try the translator of Fig. 8-4 on the example described above. A precise flow chart of the translator can be made as in Fig. 8-5, provided that the following indices are understood:

Let s index the symbols of the code being translated, from left to right.

Let S index the lines of the syntax description, from top to bottom.

Let i index the alternatives of a line, from right to left; hence S,i designates a particular alternative.

Let j index the components of an alternative, from right to left; hence S,i,j designates a particular component.

8-3. Examples of Translating ALGOL Codes into a Symbolic Machine Language

The Source and Target Languages for the Examples. Consider the translations of the simple arithmetic expression and the assignment statement of ALGOL, as the source language, into a symbolic machine language, as the target language. The target language for the purposes of our examples is described in Table 8-3 and is simply a one-address symbolic instruction system. We shall assume that a compiler has already been written that can assign addresses to a code written in this symbolic machine

TABLE 8-3. DEFINITIONS OF SYMBOLIC-MACHINE-LANGUAGE INSTRUCTIONS TO BE USED AS THE TARGET LANGUAGE FOR THE EXAMPLES†

CLA α	Replace (accumulator) by (α)
STR α	Transfer (accumulator) to α
RTP α	Raise (accumulator) to the power of (α); leave result in accumulator
MUL α	Multiply (accumulator) by (α); put product in accumulator
DIV α	Divide (accumulator) by (α); put quotient in accumulator
ADD α	Add (α) to (accumulator); put sum in accumulator
SUB α	Subtract (α) from (accumulator); put difference in accumulator

† Addresses **P**, **M**, and **Q** are each push-down lists which push down whenever an **STR** instruction is used and pop up on the use of the address by any other instruction.

TABLE 8-4. DEFINITION OF SIMPLE ARITHMETIC EXPRESSION AND ASSIGNMENT STATEMENT OF ALGOL†

	(a) *Actual terms*	(b) *Level numbers*
1	⟨var⟩ ::= **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **J** | **K** | **L**	**1** ::= **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **J** | **K** | **L**
2	⟨eq⟩ ::= **:=** / ⟨empty⟩	**2** ::= **:=** / ⟨empty⟩
3	⟨p⟩ ::= **↑** / ⟨empty⟩	**3** ::= **↑** / ⟨empty⟩
4	⟨mult op⟩ ::= **×** / **MUL** | **/** / **DIV**	**4** ::= **×** / **MUL** | **/** / **DIV**
5	⟨close⟩ ::= **)** / ⟨empty⟩	**5** ::= **)** / ⟨empty⟩
6	⟨add op⟩ ::= **+** / **ADD** | **−** / **SUB**	**6** ::= **+** / **ADD** | **−** / **SUB**
7	⟨open⟩ ::= **(** / ⟨empty⟩	**7** ::= **(** / ⟨empty⟩
8	⟨lft pt⟩ ::= ⟨var⟩⟨eq⟩ / $[C_2\}$	**8** ::= **1,2** / $[C_2\}$
9	⟨t⟩ ::= ⟨var⟩ / **CLA** $[C_1\}$ | ⟨open⟩⟨sae⟩⟨close⟩ / $\{C_2\}$	**9** ::= **1** / **CLA** $[C_1\}$ | **7,12,5** / $\{C_2\}$
10	⟨f⟩ ::= ⟨t⟩ / $\{C_1\}$ | ⟨f⟩⟨p⟩⟨t⟩ / $\{C_1\}$ **STR P** $\{C_3\}$ **RTP P**	**10** ::= **9** / $\{C_1\}$ | **10,3,9** / $\{C_1\}$ **STR P** $\{C_3\}$ **RTP P**
11	⟨mf⟩ ::= ⟨f⟩ / $\{C_1\}$ | ⟨mf⟩⟨mult op⟩⟨f⟩ / $\{C_1\}$ **STR M** $\{C_3\}$ $\{C_2]$ **M**	**11** ::= **10** / $\{C_1\}$ | **11,4,10** / $\{C_1\}$ **STR M** $\{C_3\}$ $\{C_2]$ **M**
12	⟨sae⟩ ::= ⟨mf⟩ / $\{C_1\}$ | ⟨sae⟩⟨add op⟩⟨mf⟩ / $\{C_1\}$ **STR Q** $\{C_3\}$ $\{C_2]$ **Q**	**12** ::= **11** / $\{C_1\}$ | **12,6,11** / $\{C_1\}$ **STR Q** $\{C_3\}$ $\{C_2]$ **Q**
13	⟨assignment statement⟩ ::= ⟨lft pt⟩⟨sae⟩ / $\{C_1\}$ **STR** $[C_2\}$	**13** ::= **8,12** / $\{C_1\}$ **STR** $[C_2\}$

† To be used as the source language of our examples in terms of the target language of Table 8-3. For convenience we have placed the semantics directly under the respective syntactical alternative. We have added $[C_n\}$ and $\{C_n]$ to our $\{C_n\}$ notation: the square bracket on either side indicates that the last instruction of the target-language fragment is incomplete on that side and is to be completed at a later stage. Thus, if $\{C_2]$ is **CLA A** / **ADD** _ and $[C_1\}$ is **Q**, then $\begin{matrix}\{C_2] \\ [C_1\}\end{matrix}$ becomes **CLA A** / **ADD Q**.

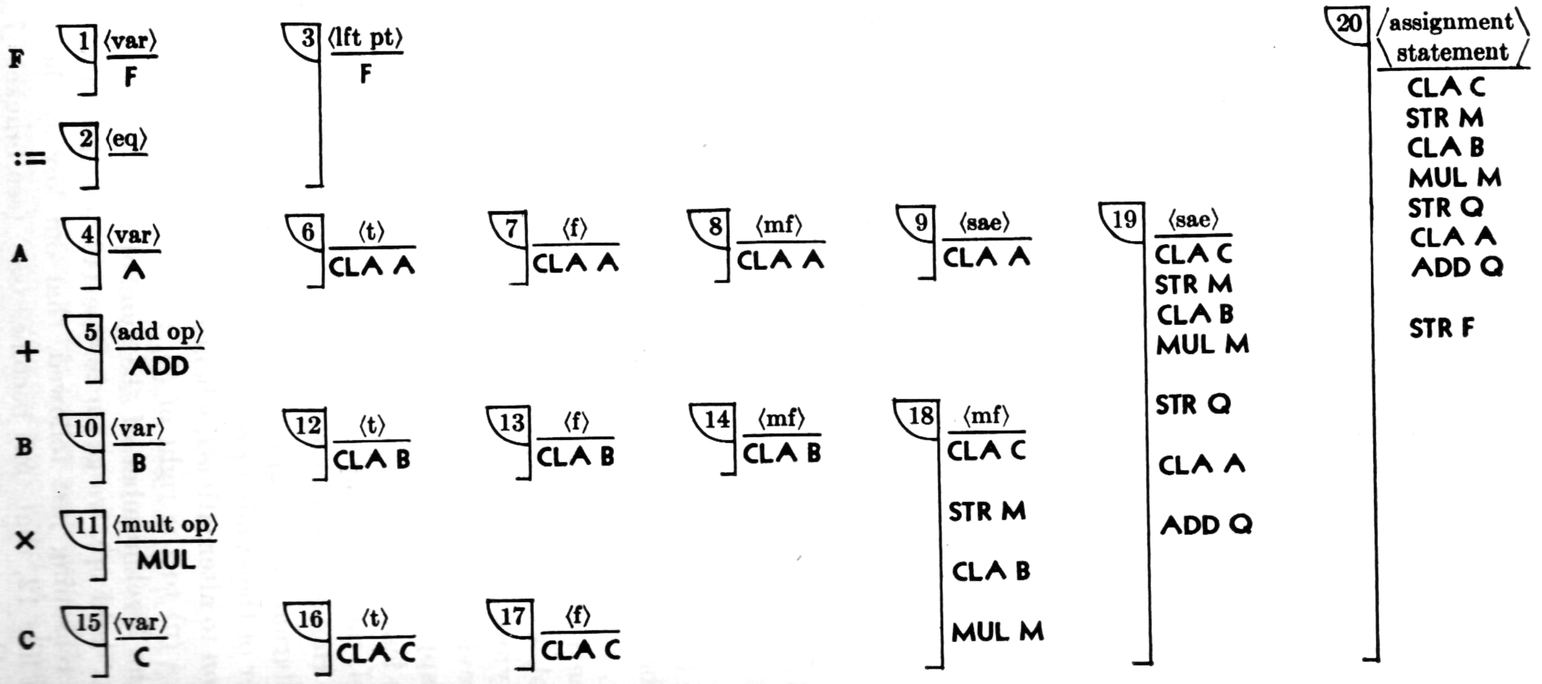
language. (See Sec. 2-6 for a discussion of symbolic coding and Sec. 5-7 for a discussion of an address-assigner program.) Hence our problem is to translate a code from the source language ALGOL into the target, symbolic machine language; this accomplished, a translated code would then have specific addresses assigned and be run on the computer.

The source language for the purposes of our examples is defined in Table 8-4, column *a*, where we have given with slight simplifications the syntax of the ALGOL simple arithmetic expression and assignment statement (see Secs. 6-3 and 6-4). In this table we have also formulated the semantics of these ALGOL definitions in terms of our illustration symbolic-machine target language. In column *a* we have placed the semantic descriptions or the target-language translated fragments *under* the corresponding syntactical alternative (rather than to the right of an ⇒ as in the above Korean translation example). In column *b* we repeat the same source language definition using level numbers. We are now ready for the translation of our first ALGOL code.

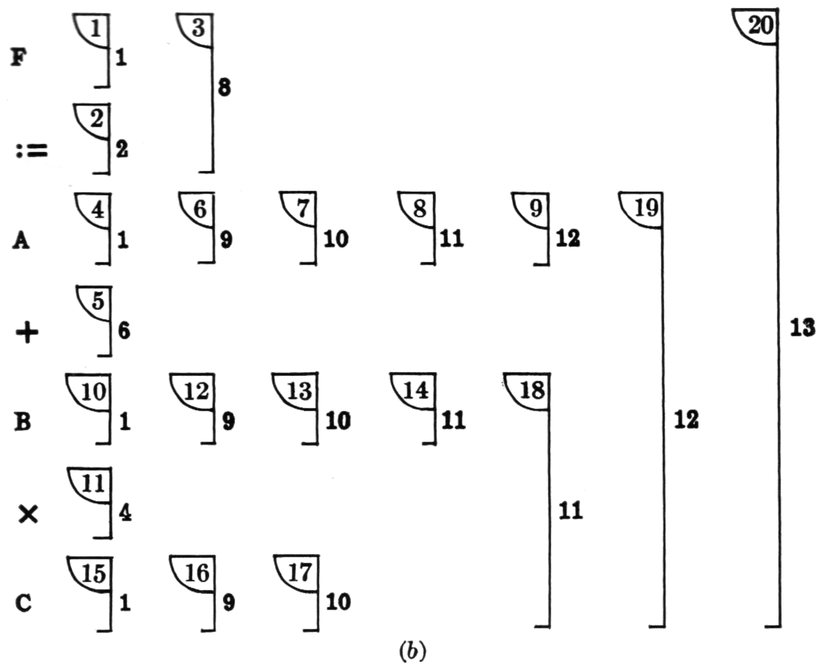
Examples of Translation. Consider the code

F:=A+B×C

Applying our translator of Fig. 8-4 (or Fig. 8-5), we find the work shown in Fig. 8-6*a*, where the numbers on the brackets indicate the order of the steps. Note from column *a* of Table 8-4 that first **F** is identified by line 1 as a ⟨var⟩. Then line 8 requires that the next symbol **:=** be examined (i.e., become the key symbol); this is identified by line 2 as ⟨eq⟩, from which, again by line 8, we obtain the ⟨lft pt⟩ of bracket 3 (see Fig. 8-6*a*). By line 13 (of Table 8-4, column *a*), the next symbol **A** becomes the key symbol, which is identified as a ⟨var⟩ by line 1; by line 8, we go on to the next symbol **+**, which is identified by line 6 as an ⟨add op⟩. By line 12, the previous symbol (namely, ⟨var⟩) becomes the key symbol. Although ⟨var⟩ appears in line 8, no match is made because the level number of ⟨eq⟩ is less than the level number of the symbol to the right of the key symbol, namely, ⟨add op⟩, and thus ⟨add op⟩ cannot be developed into ⟨eq⟩. Hence ⟨var⟩ becomes ⟨t⟩ by line 9. Looking at line 10, component 2 of alternative 1 is ⟨p⟩ with level number 3, which is less than the level number of the symbol preceding the key symbol (namely, ⟨lft pt⟩); hence we go on to alternative 2, whence ⟨t⟩ becomes ⟨f⟩. Line 10, alternative 1, requires ⟨p⟩ to the right of ⟨f⟩; but since the level number of ⟨p⟩ is less than the level number of the symbol to the right of the key symbol, namely, ⟨add op⟩, we go on to line 11. On line 11 we find, analogously to the reasoning just followed, that our key symbol ⟨f⟩ becomes ⟨mf⟩; and on line 12, that ⟨mf⟩ becomes ⟨sae⟩ (see brackets 7, 8, and 9 of Fig. 8-6*a*). However, again on line 12 of the table, we see that our key



(*a*)



(*b*)

FIG. 8-6. Successive stages in the translation of F:=A+B×C. Above the horizontal lines are shown the successive changes in the syntactical analysis, while below are shown the stages in the building up of the target-language code. The numbers on the brackets indicate the order in which the steps are made. In (*a*) we have used the actual terms; in (*b*) we have shown only the level numbers.

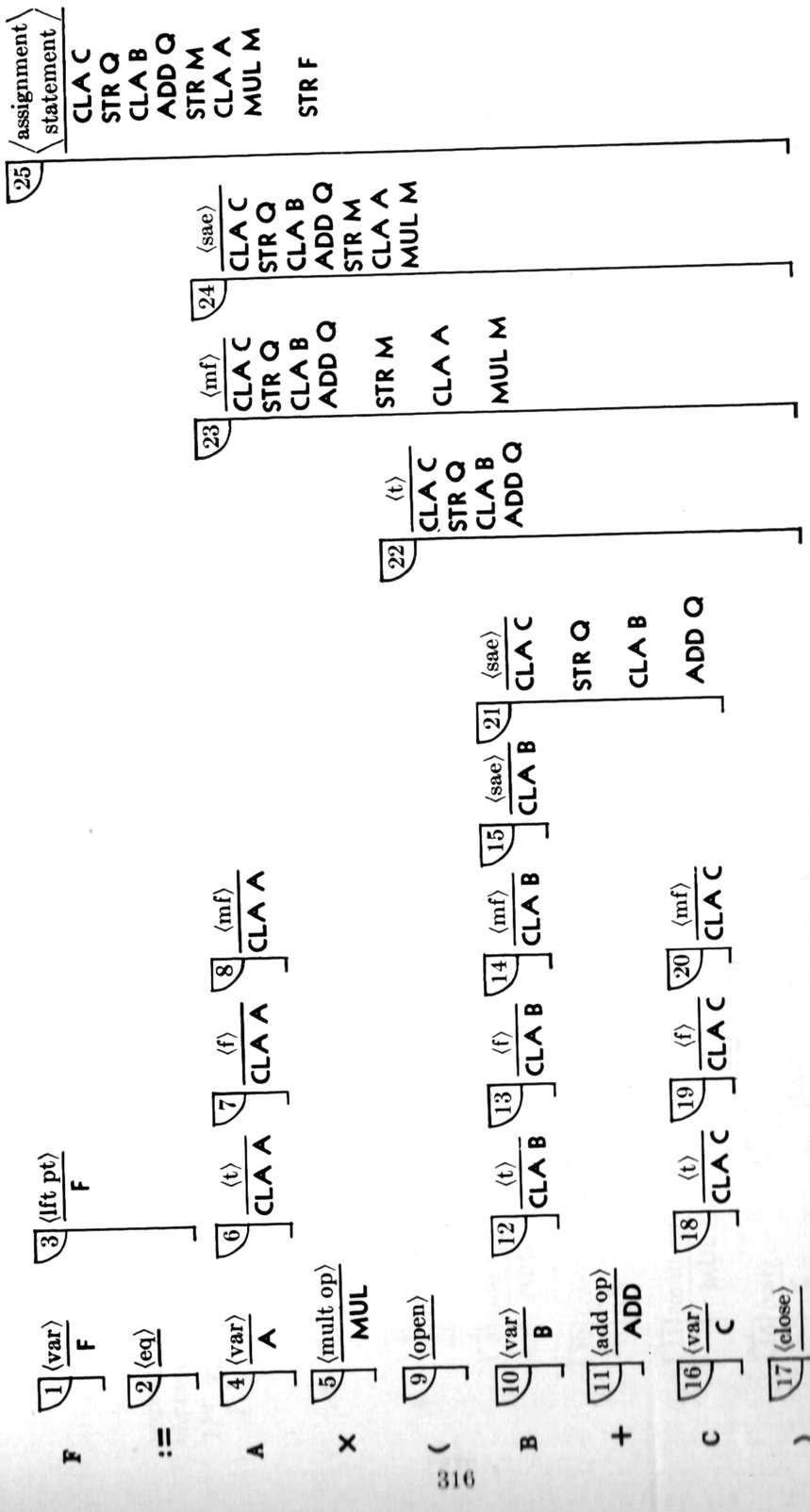


FIG. 8-7. Work for translating F:=A×(B+C).

symbol, which is now ⟨sae⟩, actually has on its right an ⟨add op⟩ as required by alternative 1, component 2; hence we choose the next symbol **B** as the new key symbol and see whether or not it can be developed into an ⟨mf⟩. It becomes first a ⟨t⟩, then an ⟨f⟩, and eventually, by line 11, an ⟨mf⟩ (see brackets 10, 11, 12, 13, and 14 of Fig. 8-6*a*). However, on line 11, alternative 1 can now be used, since ⟨mult op⟩ has already been formed (in bracket 11). Thus we choose **C** as the key symbol, which develops into an ⟨mf⟩ by absorbing the previous two symbols: i.e., by line 11, alternative 1,

⟨mf⟩ : : = ⟨mf⟩⟨mult op⟩⟨f⟩

(see bracket 18). Finally, by line 12 of the table, we combine this ⟨mf⟩ with the preceding ⟨add op⟩ and ⟨sae⟩, forming an ⟨sae⟩ in bracket 19; then by line 13 the assignment statement is formed in bracket 20. Note that we could have accomplished the entire process by utilizing the level numbers alone (see column *b* of Table 8-4 and the work in Fig. 8-6*b*).

The translation of parentheses is illustrated by

F:=A×(B+C)

the work for which is shown in Fig. 8-7. Note that the expression between the parentheses becomes an ⟨sae⟩ before the parentheses are absorbed in the formation of the ⟨t⟩.

As another example consider the translation of

F:=A↑B+C×D×E

the work for which appears in Fig. 8-8. The student should carefully check the successive steps, identifying for each the syntactical line used from Table 8-4 and noting the sections of the translator flow chart (Fig. 8-4 or 8-5) utilized. The student should also check the final code to see that it actually does compute the desired arithmetic expression. Several observations can be made about this code. First, note that the address **M** must be a push-down list, for first the code stores **(E)** in **M**, and then **(D)** is stored in **M**. Obviously **(E)** and **(D)** must be stored in different cells; however, since **(D)** is stored last, it is utilized first for multiplication, allowing **(E)** to pop up for the next multiplication (see footnote in Table 8-3). The second observation concerns the fact that the code resulting from the translation is far from minimum; it is not hard to code this function using fewer instructions. That this is true is seen, for example, in the first two instructions: first **(E)** is put into the accumulator, and then in the very next instruction it is re-stored. In order to eliminate such unnecessary instructions, we must reformulate the syntax and semantic definitions in more detail.

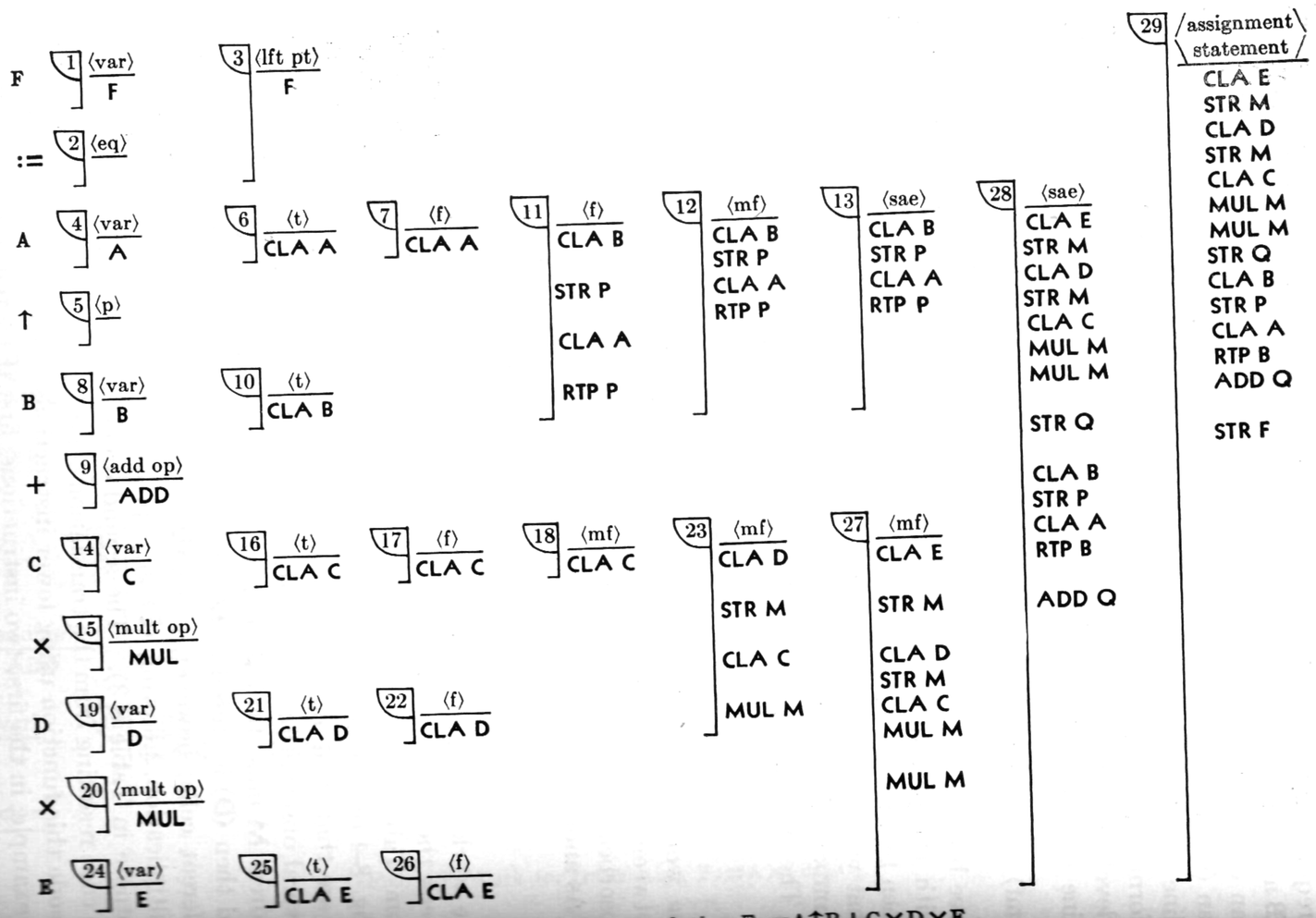


FIG. 8-8. Work for translating F:=A↑B+C×D×E.

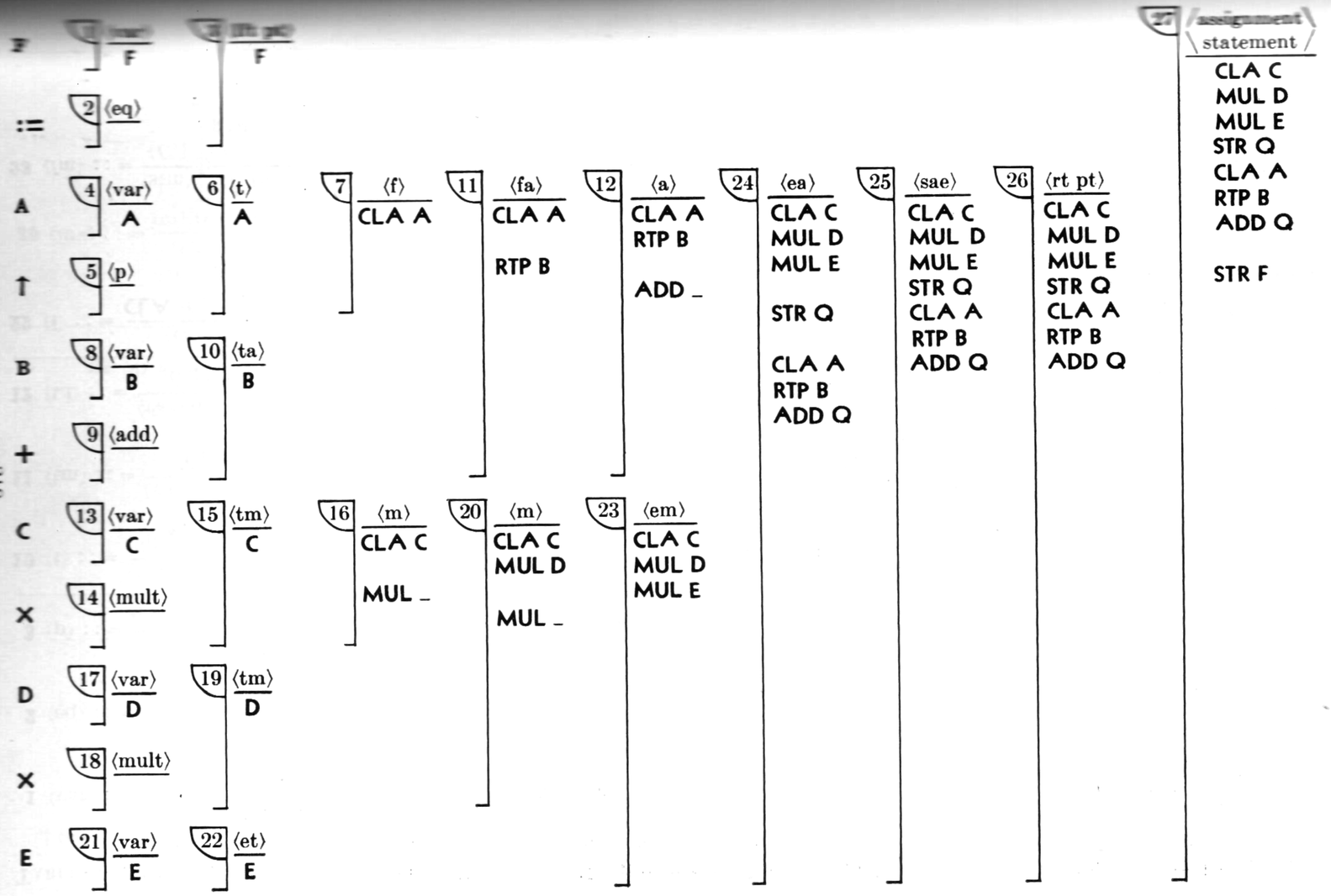


FIG. 8-9. Work for translating F:=A↑B+C×D×E utilizing the syntax and semantics of Table 8-5. Compare the resulting code with that of Fig. 8-8.

TABLE 8-5. SYNTAX AND SEMANTICS THAT RESULT IN NEAR-MINIMUM TRANSLATED CODES FOR SIMPLE ARITHMETIC EXPRESSIONS AND ASSIGNMENT STATEMENTS OF ALGOL IN TERMS OF A ONE-ADDRESS SYMBOLIC MACHINE LANGUAGE†

1 ⟨var⟩ ::= A | B | C | . . . / A | B | C | . . .

2 ⟨eq⟩ ::= := / ⟨empty⟩

3 ⟨p⟩ ::= ↑ / ⟨empty⟩

4 ⟨mult⟩ ::= × / ⟨empty⟩

5 ⟨div⟩ ::= / / ⟨empty⟩

6 ⟨add⟩ ::= + / ⟨empty⟩

7 ⟨sub⟩ ::= − / ⟨empty⟩

8 ⟨close⟩ ::=) / ⟨empty⟩

9 ⟨ve⟩ ::= ⟨open⟩⟨sae⟩⟨close⟩ / $\{C_2\}$

10 ⟨t⟩ ::= ⟨var⟩⟨p⟩ / $[C_2\}$

11 ⟨tm⟩ ::= ⟨var⟩⟨mult⟩ / $[C_2\}$

12 ⟨td⟩ ::= ⟨var⟩⟨div⟩ / $[C_2\}$

13 ⟨ta⟩ ::= ⟨var⟩⟨add⟩ / $[C_2\}$

14 ⟨ts⟩ ::= ⟨var⟩⟨sub⟩ / $[C_2\}$

15 ⟨et⟩ ::= ⟨var⟩ / $[C_1\}$

16 ⟨st⟩ ::= ⟨ve⟩⟨p⟩ / $\{C_2\}$

17 ⟨stm⟩ ::= ⟨ve⟩⟨mult⟩ / $\{C_2\}$

18 ⟨std⟩ ::= ⟨ve⟩⟨div⟩ / $\{C_2\}$

19 ⟨sta⟩ ::= ⟨ve⟩⟨add⟩ / $\{C_2\}$

20 ⟨sts⟩ ::= ⟨ve⟩⟨sub⟩ / $\{C_2\}$

21 ⟨set⟩ ::= ⟨ve⟩ / $\{C_1\}$

22 ⟨f⟩ ::= ⟨t⟩ / CLA $[C_1\}$ | ⟨st⟩ / $\{C_1\}$ | ⟨f⟩⟨t⟩ / $\{C_2\}$ RTP $[C_1\}$ | ⟨f⟩⟨st⟩ / $\{C_1\}$ STR P $\{C_2\}$ RTP P

23 ⟨fm⟩ ::= ⟨stm⟩ / $\{C_1\}$ | ⟨f⟩⟨tm⟩ / $\{C_2\}$ RTP $[C_1\}$ | ⟨f⟩⟨stm⟩ / $\{C_1\}$ STR P $\{C_2\}$ RTP P

24 ⟨fd⟩ ::= ⟨std⟩ / $\{C_1\}$ | ⟨f⟩⟨td⟩ / $\{C_2\}$ RTP $[C_1\}$ | ⟨f⟩⟨std⟩ / $\{C_1\}$ STR P $\{C_2\}$ RTP P

25 ⟨fa⟩ ::= ⟨sta⟩ / $\{C_1\}$ | ⟨f⟩⟨ta⟩ / $\{C_2\}$ RTP $[C_1\}$ | ⟨f⟩⟨sta⟩ / $\{C_1\}$ STR P $\{C_2\}$ RTP P

26 ⟨fs⟩ ::= ⟨sts⟩ / $\{C_1\}$ | ⟨f⟩⟨ts⟩ / $\{C_2\}$ RTP $[C_1\}$ | ⟨f⟩⟨sts⟩ / $\{C_1\}$ STR P $\{C_2\}$ RTP P

27 ⟨ef⟩ ::= ⟨set⟩ / $\{C_1\}$ | ⟨f⟩⟨et⟩ / $\{C_2\}$ RTP $[C_1\}$ | ⟨f⟩⟨set / $\{C_1\}$ STR P $\{C_2\}$ RTP P

28 ⟨m⟩ ::= ⟨tm⟩ / CLA $[C_1\}$ MUL _ | ⟨td⟩ / CLA $[C_1\}$ DIV _ | ⟨fm⟩ / $\{C_1\}$ MUL _ | ⟨fd⟩ / $\{C_1\}$ DIV _ | ⟨m⟩⟨tm⟩ / $\{C_2\}$ $[C_1\}$ MUL _ | ⟨m⟩⟨td⟩ / $\{C_2\}$ $[C_1\}$ DIV _ | ⟨m⟩⟨fm⟩ / $\{C_1\}$ STR M $\{C_2\}$ M MUL _ | ⟨m⟩⟨fd⟩ / $\{C_1\}$ STR M $\{C_2\}$ M DIV _

29 ⟨ma⟩ ::= ⟨m⟩⟨ta⟩ / $\{C_2\}$ $[C_1\}$ | ⟨m⟩⟨fa⟩ / $\{C_1\}$ STR M $\{C_2\}$ M

30 ⟨ms⟩ ::= ⟨m⟩⟨ts⟩ / $\{C_2\}$ $[C_1\}$ | ⟨m⟩⟨fs⟩ / $\{C_1\}$ STR M $\{C_2\}$ M

31 ⟨em⟩ ::= ⟨m⟩⟨et⟩ / $\{C_2\}$ $[C_1\}$ | ⟨m⟩⟨ef⟩ / $\{C_1\}$ STR M $\{C_2\}$ M

32 ⟨a⟩ ::= ⟨ta⟩ / CLA $[C_1\}$ ADD _ | ⟨ts⟩ / CLA $[C_1\}$ SUB _ | ⟨fa⟩ / $\{C_1\}$ ADD _ | ⟨fs⟩ / $\{C_1\}$ SUB _ | ⟨ma⟩ / $\{C_1\}$ ADD _ | ⟨ms⟩ / $\{C_1\}$ SUB _ | ⟨a⟩⟨ta⟩ / $\{C_1\}$ $[C_2\}$ ADD _ | ⟨a⟩⟨ts⟩ / $\{C_1\}$ $[C_2\}$ SUB _ | ⟨a⟩⟨fa⟩ / $\{C_1\}$ STR Q $\{C_2\}$ Q ADD _ | ⟨a⟩⟨fs⟩ / $\{C_1\}$ STR Q $\{C_2\}$ Q SUB _ | ⟨a⟩⟨ma⟩ / $\{C_1\}$ STR Q $\{C_2\}$ Q ADD _ | ⟨a⟩⟨ms⟩ / $\{C_1\}$ STR Q $\{C_2\}$ Q SUB _

33 ⟨ea⟩ ::= ⟨a⟩⟨et⟩ / $\{C_2\}$ $[C_1\}$ | ⟨a⟩⟨ef⟩ / $\{C_1\}$ STR Q $\{C_2\}$ Q | ⟨a⟩⟨em⟩ / $\{C_1\}$ STR Q $\{C_2\}$ Q

34 ⟨open⟩ ::= (/ ⟨empty⟩

35 ⟨sae⟩ ::= ⟨et⟩ / $\{C_1\}$ | ⟨ef⟩ / $\{C_1\}$ | ⟨em⟩ / $\{C_1\}$ | ⟨ea⟩ / $\{C_1\}$

36 ⟨rt pt⟩ ::= ⟨sae⟩ / $\{C_1\}$ | ⟨open⟩⟨sae⟩⟨ve⟩ / $\{C_2\}$

37 ⟨lft pt⟩ ::= ⟨var⟩⟨eq⟩ / $[C_2\}$

38 ⟨assignment statement⟩ ::= ⟨lft pt⟩⟨rt pt⟩ / $\{C_1\}$ STR $[C_2\}$

† See Table 8-3.

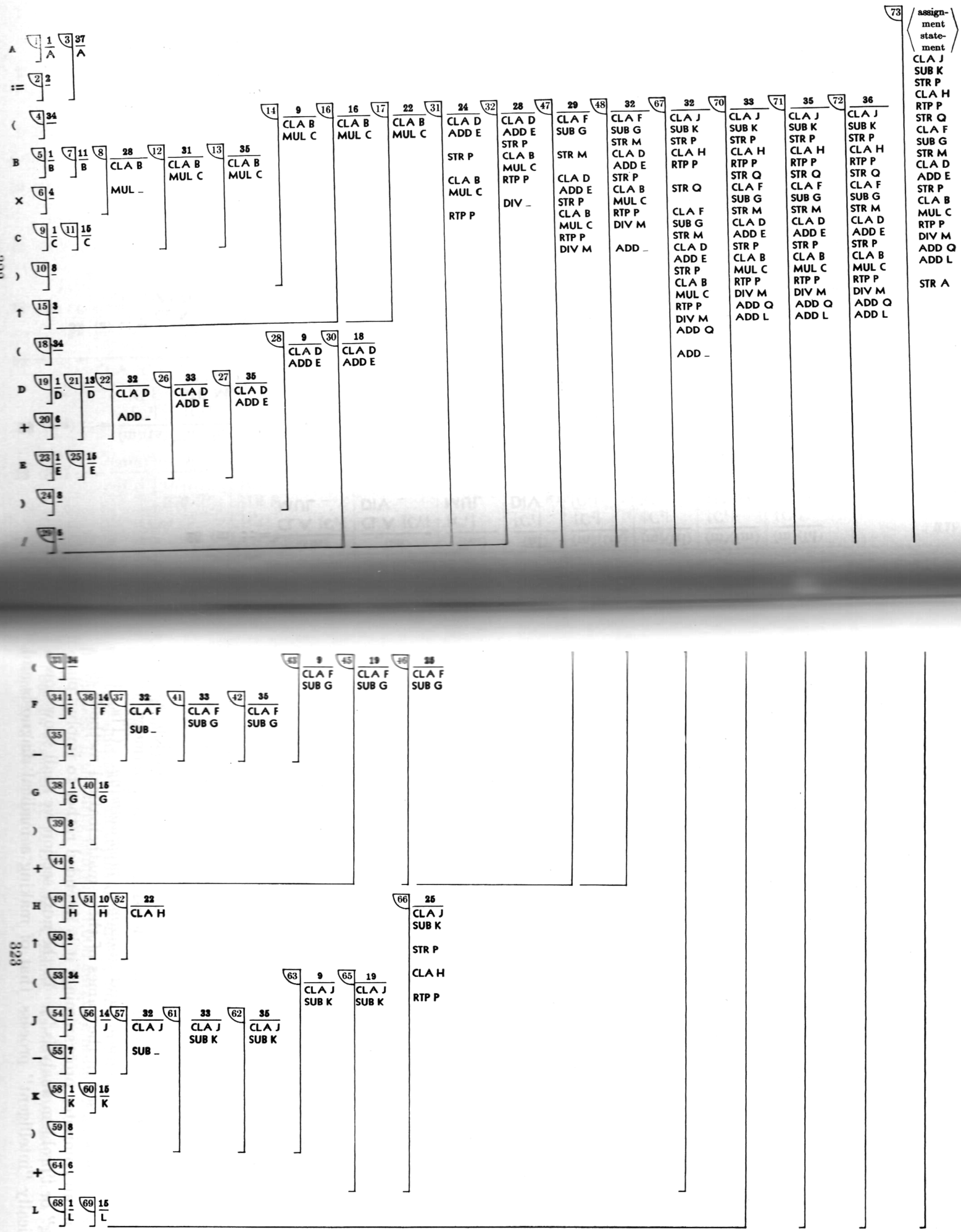


FIG. 8-10. Work for translating **A:=(B×C)↑(D+E)/(F−G)+H↑(J−K)+L** utilizing the syntax and semantics of Table 8-5.

Example of a More Efficient Syntax and Semantic Description. In Table 8-5, we present a collection of syntax and semantic descriptions that will produce a near-minimum code. For example, in Fig. 8-9 we have shown the work for translating the same assignment statement as in Fig. 8-8, namely **F:=A↑B+C×D×E**; now only 8 instead of 14 instructions are required. In Fig. 8-10 we have illustrated the work for translating

$$\mathbf{A:=(B\times C)\uparrow(D+E)/(F-G)+H\uparrow(J-K)+L}$$

The reader will benefit greatly by first observing the final code and how it builds up the desired function, and then studying the work in Fig. 8-10 and how the translator builds up this final code. Although we have written the syntax in terms of syntactical entities for ease of study, we have given the example of Fig. 8-10 in terms of level numbers for ease in tracing. In building up the translated code in this near-minimum fashion, the translator almost appears to be "intelligent." This behavior of the translator, however, is clearly entirely dependent on the "efficiency" of the syntactical and semantic description.

8-4. Deductive Inference: Mathematical Optimization

Modern Mathematical Decision Theories. Planning, controlling, adapting, self-regulating, optimizing—all these describe processes which are attributes of "intelligent" systems, and programming a computer to aid in such processes can be interpreted as imparting a measure of "intelligence" to the computer. In this and the next section we shall be concerned with elementary illustrations of how a computer can be made to exhibit such attributes through *deductive inference.* We classify modern mathematical decision theories as deductive-inference methodology because these mathematical methods are the outcome of or conclusions drawn through logical (mathematical) deduction made from given premises; the methods produce optimum or measured near-optimum solutions and are *consequences of mathematical results.* We use the term deductive inference, as opposed to inductive inference, to describe intelligent behavior based upon deductive mathematical optimizing or regulating methods, as opposed to hypothesis making and intelligent trial and error (which are the subjects of Sec. 8-6).

To illustrate these points, we shall consider in this section selected examples of modern mathematical decision theories, namely, *linear programming, game theory,* and *dynamic programming.* (The term "programming" used in the names of these mathematical methods does not mean computer programming; it is intended rather to imply "planning" or "problem analysis.") In the next section we shall show how a typically "intelligent" process, that of making a medical diagnosis and

treatment plan, can be formulated in terms of an integrated sequence of such mathematical decision-making methods—i.e., in terms of deductive inference.

The reader may wonder why we consider what are primarily *mathematical* methods in this chapter on *programming* to achieve artificial intelligence. The reason is, of course, that whatever measure of intelligence is achieved by a computer arises from mathematical procedures carried out by the programs and routines inside the computer. Actually, the role of the computer is twofold: first, the fact that a computer is to be utilized forces the precise and unambiguous analysis of the approach—covering all alternatives and eventualities—so that a "programmable" process results; and second, the availability of the computer makes feasible approaches to problems that otherwise would never have been attempted.

Linear Programming: Decision under Certainty. Decision processes are characteristically associated with three types of ingredients: the list of possible *states* of the system, the list of *alternative decisions* that can be made affecting the system, and the collection of *values* which is associated with each decision-state pair. When the state of the system is known, we say that the problem is one of decision under *certainty;* when only the probability is known of the system being in each of the possible states, we call the problem one of decision under *risk;* and when *no* knowledge is available about which state the system is in, the problem is known as one of decision under *uncertainty.*† Consider first the simplest example of a problem of decision under certainty. Suppose that a patient has disease d_j and that there are available two alternative treatments, T_1 and T_2, with prognosis related values V_{1j} and V_{2j}. The situation can be represented by the following array:

		d_j	*System state*
Alternative decisions	T_1	V_{1j}	*Decision-state values*
	T_2	V_{2j}	

Trivially, whichever treatment corresponds to the greatest value is chosen.

A more complicated decision problem under certainty can involve *linear programming.* For example, suppose that there were two possible treatments for a cancer patient, drug administration and radiation therapy, and that, in terms of some common units of effectiveness (e.g., by bioassay), the drug contains 1,000 curative units/oz and the radiation

† See, for example, R. D. Luce and H. Raiffa, "Games and Decisions," John Wiley & Sons, Inc., New York, 1957.

gives 1,000 curative units/min. Suppose the patient must receive at least 3,000 curative units of treatment. However, both treatments are toxic to the patient so that, again in some common units, the drug contains 400 toxic units/oz, while the radiation produces 1,000 toxic units/min; suppose the patient must not receive more than 2,000 toxic units. Finally, in terms of discomfort experienced by the patient, suppose that the drug was three times as noxious per ounce as the radiation per minute. The problem is to determine the doses that will satisfy the curative and toxic criteria of the treatment but will *minimize* the discomfort to the patient. If X_1 is the number of ounces of the drug to be prescribed and X_2 is the number of minutes of radiation to be received, then to cure the patient we must have

$$1{,}000X_1 + 1{,}000X_2 \geq 3{,}000$$

but because of the toxic effects,

$$400X_1 + 1{,}000X_2 \leq 2{,}000$$

The problem is to determine X_1 and X_2 that satisfy these inequalities and minimize the "discomfort"

$$D = 3X_1 + X_2$$

Since in addition $X_1 \geq 0$ and $X_2 \geq 0$, the solution lies in the crosshatched area of Fig. 8-11. However, this last equation is the equation of a plane, the points of which give the value of the discomfort D for every pair of values X_1, X_2. Clearly the minimum value of D (consistent with the restrictions on X_1 and X_2) must lie on one of the vertices of the triangular area on the D plane whose projection is the crosshatched area on the X_1X_2 plane. Hence check the value of $3X_1 + X_2$ at each of the three points (3,0), (5,0), and ($\frac{5}{3}$,$\frac{4}{3}$), finding D to be, respectively, 9, 15, and 6.3. Thus combined treatment is indicated at $\frac{5}{3}$ oz of drug and $\frac{4}{3}$ min of radiation. The problem we have been solving is of course manifestly unrealistic, but only because of the oversimplification inherent in any such illustrative example.

Other, more realistic problems involve a multitude of factors in many inequalities. Such problems may involve optimization (or minimization) of factors associated with the organization of industrial economies, planning optimum military-defense strategies, or choosing a reducing diet. As we have just illustrated, the linear-programming problem involved is the optimization of a linear function subject to linear constraints. In general, realistic problems are so large that the use of computers is required to make their solution feasible.

Game Theory: Decision under Uncertainty. If there are several possible states in which the system may be, say $d_1, d_2, \ldots, d_n$ with respective

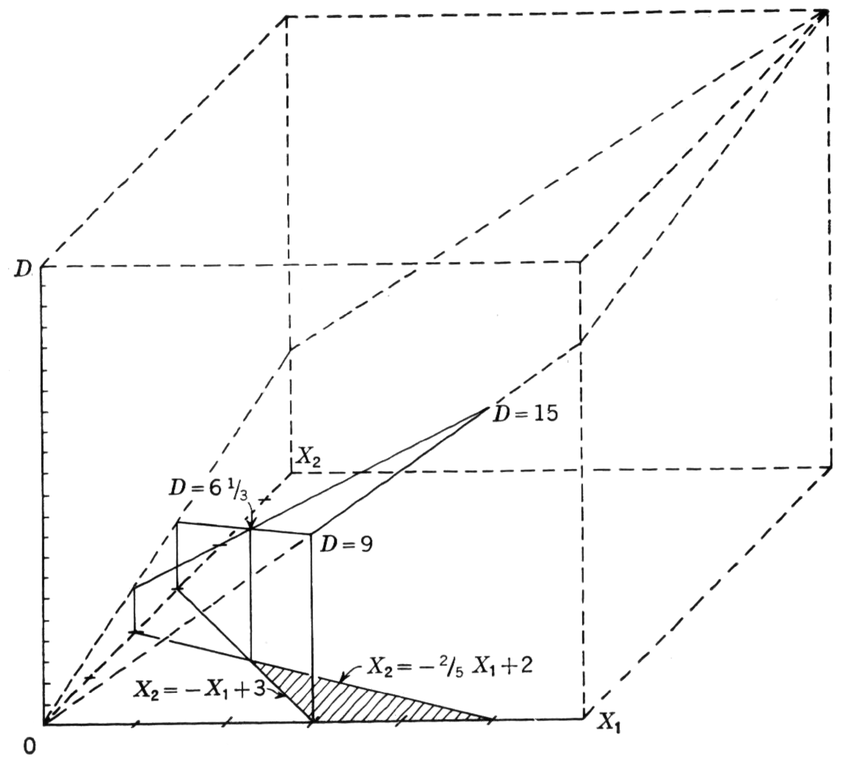


FIG. 8-11. Solution to the drug-radiation linear-programming problem.

probabilities $P_1, P_2, \ldots, P_n$; if the alternative decisions are $T_1, T_2, \ldots, T_m$; and if V_{ij} is the value of decision T_i when the system is in state d_j, then we can make the following array:

	d_1	d_2	$\cdots$	d_n
	P_1	P_2	$\cdots$	P_n
T_1	V_{11}	V_{12}	$\cdots$	V_{1n}
T_2	V_{21}	V_{22}	$\cdots$	V_{2n}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
T_m	V_{m1}	V_{m2}	$\cdots$	V_{mn}

The choice of a decision that will maximize the resulting "expected" value is found by computing

$$E_i = \sum_{j=1}^{n} V_{ij}P_j \tag{8-1}$$

for each decision T_i and choosing that decision with the greatest E_i. For instance, consider 7,000 patients for all of whom two tentative possible diagnoses (d_1 or d_2) have been made with probabilities $\frac{5}{7}$ and $\frac{2}{7}$, respectively. Suppose that there exists a treatment T_1 which is 90 per cent effective against disease d_1 and only 30 per cent effective against d_2,

that is, $V_{11} = 0.9$ and $V_{12} = 0.3$, a treatment T_2 which is 10 per cent effective against d_1 but 100 per cent effective against d_2, that is, $V_{21} = 0.1$ and $V_{22} = 1.0$ (see Fig. 8-12*a*). Under which treatment would we expect to cure the most patients? Simple substitution in Eq. (8-1) shows that

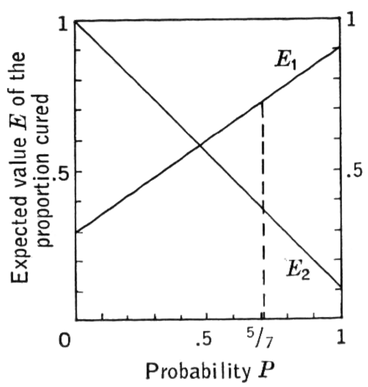
$$E_1 = {}^{51}\!/_{70} \qquad \text{and} \qquad E_2 = {}^{25}\!/_{70}$$

and hence we would expect to cure more patients with treatment T_1. However, for any particular patient we do not know which of the two diseases he has, and therefore the situation is one of *risk* for that patient, even though the treatment of choice will maximize the probability of curing each patient. For two diseases, in general, if we let P be the

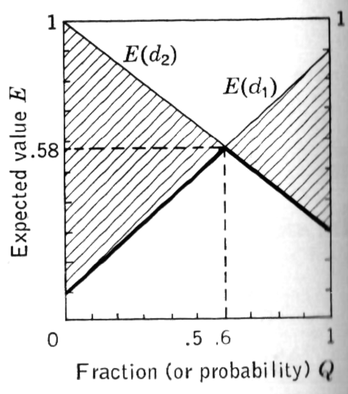
	d_1	d_2
T_1	0.9	0.3
T_2	0.1	1.0

Values

(*a*)



(*b*)



(*c*)

FIG. 8-12. Illustration of solution to problem with values as in (*a*), under (*b*) risk and (*c*) uncertainty.

probability that a patient has d_1, then $1 - P$ will be the probability that he has d_2, and using the same values of our example, we can write the expected values from Eq. (8-1) as

$$E_1 = 0.9P + 0.3(1 - P)$$
$$E_2 = 0.1P + (1 - P)$$

Figure 8-12*b* illustrates the graphs of these two equations (with the point for the given $P = {}^5\!/_7$ indicated). Note that T_1 is the treatment of choice for P to the right of where the lines cross and T_2 is the treatment of choice for P to the left of the crossing.

Suppose next that the probabilities P_i were not known, i.e., that there were *no information available* concerning the probabilities; this is the case of *uncertainty*, and the process of choosing the optimum decision is called *game theory*. In the terminology of game theory we can recast the above example of treatment choice as follows: There are two "players," the physician and nature. The physician is trying to determine the best strategy, i.e., choice of treatment (the decision), from his limited knowl-

edge of nature. The matrix representation of values, given in, say, Fig. 8-12*a*, constitutes the payoffs—what the physician will "win" and nature "lose." But this time we do *not* know the probabilities that the 7,000 patients have d_1 or d_2. Our problem is to choose that treatment which will ensure that we cure the largest number of people—that is, *to maximize the minimum possible number of patients that we expect can be cured.* There are actually three ways we can choose the treatment: (1) treat all patients with T_1, (2) treat all patients with T_2, and (3) treat some patients with T_1 and others with T_2. The first two ways are called "pure strategies," the third, a "mixed strategy."

Consider the values of Fig. 8-12*a*, and suppose we choose the third method of treatment (which really includes the first two anyway). Let Q be the fraction of patients to be treated with T_1; then $1 - Q$ is the fraction to be treated with T_2. Observe that, if all the patients had d_1, we would expect to cure a proportion of patients

$$E(d_1) = 0.9Q + 0.1(1 - Q)$$

We have graphed this equation in Fig. 8-12*c*. Similarly, if all the patients had d_2, we would expect to cure a proportion of patients

$$E(d_2) = 0.3Q + 1.0(1 - Q)$$

We have also graphed this equation in Fig. 8-12*c*. If some of the patients had d_1 and the rest d_2, then the expected value would fall inside the hatched area for any particular value of Q. Evidently the lower (thickened) line in Fig. 8-12*c* represents the minimum proportion of patients that we can certainly expect to cure not knowing how many have d_1 or d_2. For $Q = 0.6$, this minimum number is a maximum, and we would expect to cure 58 per cent of the patients. Hence 60 per cent of the patients should be treated with T_1, and the remaining 40 per cent with T_2.

To arrange for such a treatment is easy: Separate the patients at random into two groups, one containing 60 per cent of the patients, the other containing 40 per cent; the former group will be given T_1, the latter T_2. However, there is another way of arranging for such a treatment, by choosing a random number as each patient comes up for treatment. In fact, for the physician with a single patient, the method of choosing a random number must be used, for the patient cannot be divided up 60-40 for treatment! In this manner the concept of mixed strategy can be applied to a single case.

Such a method for choosing the treatment may be very hard to appreciate at first contact, but this is just the method used every day when probabilities are applied to single situations. Of course, in actual practice some further information bearing on the choice of treatment would be

sought—i.e., the formulation of the problem of which treatment to give the patient is far more complicated than that posed by the single problem discussed above. In conclusion we quote J. D. Williams† on the role of game theory:

While there are specific applications today, despite the current limitations of the theory, perhaps its greatest contribution so far has been an intangible one: the general orientation given to people who are faced with overcomplex problems. Even though these problems are not strictly solvable—it helps to have a framework in which to work on them. The concepts of a strategy, the representation of the payoffs, the concepts of pure and mixed strategies, and so on, give valuable orientation to persons who must think about complicated situations.

Dynamic Programming. Consider next the problem of making a *sequence of decisions* so as to maximize some value after, say, n such decisions. A commonly used example for introducing the concepts involved in this type of problem is that of the toymaker's dilemma. Suppose a toymaker can find himself in one of four states: manufacturing a toy that (1) is a best seller, (2) is a good seller, (3) is a poor seller, and (4) does not sell at all. Let us denote these states by H_1, H_2, H_3, and H_4. Now suppose that if the toymaker is in any state, say H_i, then there is a *transition probability* p_{ij} that in the next month he will be in state H_j, or p_{ii} that he will remain in state H_i. Associated with each such transition there is a *value*, or profit, that can be expected, say v_{ij} for the value of the transition from state H_i to state H_j, or v_{ii} for remaining in state H_i. However, notwithstanding the fickleness of the buying public, the toymaker does have some control over the situation: suppose, for instance, that for each state he can make a decision between two alternative courses of action that can change the transition probabilities and associated values. For example, if he has a best-seller toy, he could advertise to keep the toy in the public favor, or he could just rest on his laurels; the profit would be less in the former case than the latter owing to the advertising costs, but the probability of staying in the best-selling state would be greater in the former than the latter case. If he has a poor toy, he could invest in research to develop a better toy or he could just hope that the toy fashions will change next month: in the former case his probability for a transition to a better state would be higher, but he might have to use up reserves for funding the research; in the latter case he would not be increasing the probability for a transition to a better state, but he would not have to dig into his reserves.

Let T_1 and T_2 be the two courses of action that can be taken in any state; let p_{ij}^k and v_{ij}^k be the transition probability and the associated value

† J. D. Williams, "The Compleat Strategyst," p. 217, McGraw-Hill Book Company, Inc., New York, 1954.

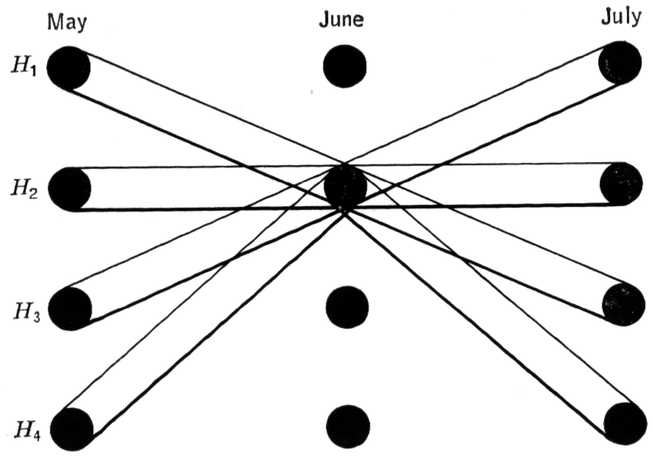


FIG. 8-13. Illustration of transitions among states: the light lines represent decision T_1, the heavy lines decision T_2; transitions are shown *to* state H_2 in June from all possible states in May and *from* state H_2 in June to all possible states in July.

TABLE 8-6. ARRAY OF TRANSITION PROBABILITIES AND VALUES, WITH ILLUSTRATIVE MAGNITUDES

State	Decision	(a) Probabilities and values	(b) Illustrative values used in the example in the text
H_1 best seller	T_1	p_{11}^1 p_{12}^1 p_{13}^1 p_{14}^1 v_{11}^1 v_{12}^1 v_{13}^1 v_{14}^1	0.5 0.3 0.1 0.1 500 300 100 −200
	T_2	p_{11}^2 p_{12}^2 p_{13}^2 p_{14}^2 v_{11}^2 v_{12}^2 v_{13}^2 v_{14}^2	0.2 0.4 0.3 0.1 700 500 300 −100
H_2 good seller	T_1	p_{21}^1 p_{22}^1 p_{23}^1 p_{24}^1 v_{21}^1 v_{22}^1 v_{23}^1 v_{24}^1	0.4 0.3 0.2 0.1 300 200 50 −200
	T_2	p_{21}^2 p_{22}^2 p_{23}^2 p_{24}^2 v_{21}^2 v_{22}^2 v_{23}^2 v_{24}^2	0.2 0.3 0.4 0.1 600 400 100 −200
H_3 poor seller	T_1	p_{31}^1 p_{32}^1 p_{33}^1 p_{34}^1 v_{31}^1 v_{32}^1 v_{33}^1 v_{34}^1	0.2 0.3 0.3 0.2 300 100 0 −300
	T_2	p_{31}^2 p_{32}^2 p_{33}^2 p_{34}^2 v_{31}^2 v_{32}^2 v_{33}^2 v_{34}^2	0.1 0.1 0.5 0.3 600 400 100 −200
H_4 no sales	T_1	p_{41}^1 p_{42}^1 p_{43}^1 p_{44}^1 v_{41}^1 v_{42}^1 v_{43}^1 v_{44}^1	0.1 0.2 0.4 0.3 300 0 −100 −400
	T_2	p_{41}^2 p_{42}^2 p_{43}^2 p_{44}^2 v_{41}^2 v_{42}^2 v_{43}^2 v_{44}^2	0.1 0.1 0.2 0.6 500 300 0 −200

of going from state i to state j for decision k on the alternative courses of action. Then, for instance, (p^1_{12},v^1_{12}), (p^1_{22},v^1_{22}), (p^1_{32},v^1_{32}), and (p^1_{42},v^1_{42}) represent the possible transitions *to* state H_2 under decision T_1 from the previous month, and (p^1_{21},v^1_{21}), (p^1_{22},v^1_{22}), (p^1_{23},v^1_{23}), and (p^1_{24},v^1_{24}) represent the possible transitions *from* state H_2 under decision T_1 for the following month (see Fig. 8-13). The situation is summarized in Table 8-6, column *a*.

We are now ready to state the toymaker's problem: How can he plan a sequence of decisions to maximize his expected profit (value) in n months? The solution to this problem will take the form of an array giving the best decision for each possible state in which the toymaker may find himself and for every month. For instance, Fig. 8-14 shows a 6-month solution to our problem for the transition probabilities and values given in Table 8-6, column *b*. Thus, in order to maximize his chances of obtaining the largest total profit by the end of September, the toymaker need merely observe his state and the month and read from Fig. 8-14 the best decision to make.

	April	May	June	July	August	September
H_1	T_1	T_1	T_1	T_1	T_2	T_2
H_2	T_1	T_1	T_2	T_2	T_2	T_2
H_3	T_1	T_1	T_1	T_1	T_1	T_2
H_4	T_1	T_1	T_1	T_2	T_2	T_2

FIG. 8-14. Solution to toymaker's problem for transition probabilities and values shown in Table 8-6, column *b*.

So far we have stated the problem and indicated the form of the solution. We now show how the solution is obtained. First, note that if the toymaker were in state H_i and were interested only in maximizing his immediate expected profit, i.e., for one particular month, then the decision would be made by calculating

$$E_i^1 = \sum_j p^1_{ij}v^1_{ij} = p^1_{i1}v^1_{i1} + p^1_{i2}v^1_{i2} + p^1_{i3}v^1_{i3} + p^1_{i4}v^1_{i4}$$

and

$$E_i^2 = \sum_j p^2_{ij}v^2_{ij} = p^2_{i1}v^2_{i1} + p^2_{i2}v^2_{i2} + p^2_{i3}v^2_{i3} + p^2_{i4}v^2_{i4}$$

and choosing decision T_1 if E_i^1 were the larger and T_2 if E_i^2 were the larger. However, in the long run this might not optimize the total profit. For example, if the toymaker were in the poor state, and it were still early in the game, then in the long run it might pay to invest in research to develop a better toy and lose money for a month or two with higher prospects of having a best seller later on, rather than to do nothing with no increasing prospects of having a better toy.

We solve this problem by means of Richard Bellman's *principle of optimality*, which follows: Let $V_i(n)$ be the *total* expected value obtained in n months, starting from H_i, if an *optimum sequence of decisions is chosen* at each month. Then the principle of optimality states that

$$V_i(n-1) = \max_k \sum_j p^k_{ij}[v^k_{ij} + V_j(n)]$$
$$= \max_k \left[E_i^k + \sum_j p^k_{ij}V_j(n) \right]$$

Thus if an optimum sequence of decisions has already been chosen for the final n months, then the principle describes how to choose the optimum decision for the *previous* month. The technique for performing the calculations is as follows: First, choose the boundary, or "end," conditions $V_j(N)$, if N months in all are to be planned for. Next, for each possible state i and alternative decision k, compute

$$\sum_j p^k_{ij}[v^k_{ij} + V_j(N)] = E_i^k + \sum_j p^k_{ij}V_j(N)$$

and for each i note for which k a maximum is obtained; call this $V_i(N-1)$. Then compute

$$\sum_j p^k_{ij}[v^k_{ij} + V_j(N-1)] = E_i^k + \sum_j p^k_{ij}V_j(N-1)$$

and again for each i note for which k a maximum is obtained, calling this $V_i(N-2)$; and so forth. Of course, the k indicates each time which decision is to be made. In this way the sequence of decisions of Fig. 8-14 was computed, for boundary conditions

$$V_1(6) = V_2(6) = V_3(6) = V_4(6) = 0$$

The iterative technique utilized to obtain the solution of the dynamic-programming problem just described is typical of a large class of problems involving over-all rather than local optimization, control analysis, and planning in the large range rather than in the neighborhood. We quote Richard Bellman:†

> The last years have seen an extraordinary acceleration of interest in the analysis and control of systems with a particular focussing of attention on self-regulating systems. We must face the fact that we have landed with both feet in the era of automation. . . . We must then develop new techniques in order to obtain solutions to mathematical problems posed by control processes. Fortunately, to help us in our efforts, we possess a new tool, the digital computer, whose existence changes the very concept of the term "solution."

† R. Bellman, "Adaptive Control Processes," pp. 3, 7, Princeton University Press, Princeton, N.J., 1961.

8-5. Deductive Inference: Programming for Medical Diagnosis†

Reasoning in Medical Diagnosis. The purpose of this section is to illustrate the analysis of a complicated reasoning process—namely, the reasoning inherent in a medical diagnosis—in terms of the methods of deductive inference. The reasoning foundations of medical diagnosis have received recent emphasis as the result of the increasing interest in the use of computers to aid the diagnostic processes. Before computers can be used effectively for such purposes, however, we need to know more about how the physician makes a medical diagnosis.

If a physician is asked, "How do you make a medical diagnosis?" his explanation of the process might be as follows: "First, I obtain the case facts from the patient's history, physical examination, and laboratory tests. Second, I evaluate the relative importance of the different signs and symptoms. Some of the data may be of first-order importance and other data of less importance. Third, to make a differential diagnosis, I list all the diseases which the specific case can reasonably resemble. Then I exclude one disease after another from the list until it becomes apparent that the case can be fitted into a definite disease category, or that it may be one of several possible diseases, or else that its exact nature cannot be determined." This, obviously, is a greatly simplified explanation of the process of diagnosis, for the physician might also comment that after seeing a patient he often has a "feeling about the case." This "feeling," although hard to explain, may be a summation of his impressions concerning the way the data seems to fit together, the patient's reliability, general appearance, facial expression, and so forth; and the physician might add that such thoughts do influence the considered diagnosis. No one can doubt that complex reasoning processes are involved in making a medical diagnosis. The diagnosis is important because it helps the physician to choose an optimum therapy, a decision which in itself demands another complex reasoning process.

Our goal is the interpretation of medical diagnosis in terms of a mathematical formulation that accurately reflects the characteristics of medical diagnostic processes. Such a formulation of medical diagnosis employs a sequential decision technique, using logical analysis to determine possible alternative diagnoses, probabilistic analysis to determine the probabilities of these alternatives, and then value theory and dynamic programming

† This section is based on the following articles by R. S. Ledley and L. B. Lusted: Reasoning Foundations of Medical Diagnosis, *Science*, vol. 130, no. 3366, pp. 9–21, July 3, 1959; The Use of Electronic Computers in Medical Data Processing, *IRE Trans. on Med. Electronics*, vol. ME-7, pp. 31–47, January, 1960; and Medical Diagnosis and Modern Decision Making, *Proc. Appl. Math. Symposium*, March, 1961, American Mathematical Society, Providence, R.I., 1962.

to assist in the choice of treatment plan. In practice, the physician first utilizes an iterative procedure of logical analysis and diagnostic testing, employing in each cycle tests of increasingly specific nature for further narrowing the alternative possible diagnoses. When alternatives still remain, Bayes' formula presents important quantitative insight into the relative roles played in the diagnostic decision by both the disease-symptom relationships of medical knowledge and the circumstantial factors involved in the location, the season, epidemics, etc. Almost surely the primary purpose for making a diagnosis is to guide the processes of formulating the treatment plan. In many cases the treatment plan must be formulated in the absence of a precisely defined diagnosis, and a diagnostic-analysis treatment-formulation iterative cycle, in the dynamic-programming sense, can be utilized. If such procedures are to be applied in practical situations, the use of high-speed electronic computers is almost certainly indicated; but even in using such computers, care must be taken that the computational techniques employed can handle the data manipulation within a feasible time period.

Listing the Diagnostic Possibilities. There are three ingredients to the logical concepts inherent in medical diagnosis; these are (1) medical knowledge, (2) the signs and symptoms presented by the patient, (3) the final medical diagnosis itself. Medical knowledge presents certain information about relationships that exist between the symptoms and the diseases. The symptoms† present further information associated with this particular patient. With these two sources of available information, and by means of the propositional calculus of symbolic logic, the diagnosis is made. Consider n symptoms, denoted by

$$S(1),\ S(2),\ \ldots,\ S(n)$$

and m diseases, denoted by

$$D(1),\ D(2),\ \ldots,\ D(m)$$

The notation to be used here for negation (*not*), logical product (*and*), logical sum (*inclusive or*), and implies (*if, then*) is that used in Chap. 10. A listing can be made of all the 2^{n+m} combinations of symptoms and diseases a patient can and cannot have. For example, the columns of Fig. 8-15 represent such a listing for three symptoms and three diseases, with a unit signifying that the patient has the symptom or disease of the corresponding row, a zero that he does not. The column with the arrow represents a patient having symptoms $S(1)$, having $S(2)$, *not* having $S(3)$, *not* having $D(1)$, having $D(2)$, and *not* having $D(3)$. Each such

† For the purposes of this section, the terms *symptoms*, *signs*, and *tests* are considered synonymous and include medical history, laboratory-test results, etc. The physician must determine whether a symptom exists or not.

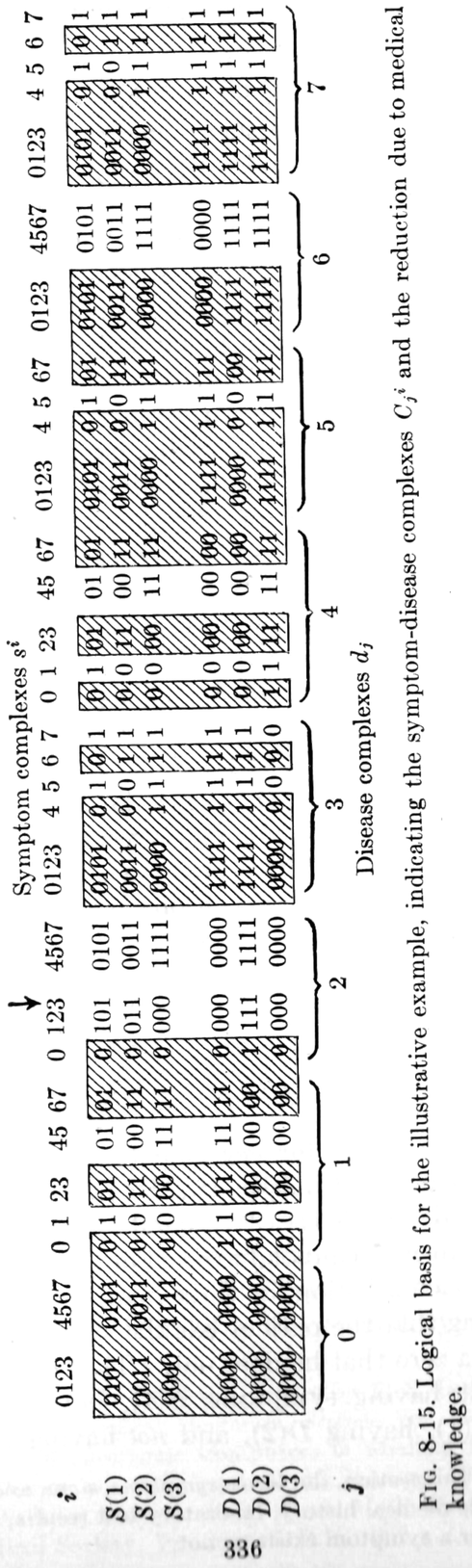


FIG. 8-15. Logical basis for the illustrative example, indicating the symptom-disease complexes C_j^i and the reduction due to medical knowledge.

combination of symptoms and diseases, called a *symptom-disease complex* (henceforth, sdc), will be denoted by C_j^i; each combination of symptoms, a *symptom complex* (sc), will be denoted by s^i; and each combination of diseases, a *disease complex* (dc), by d_j. This notation is illustrated in Fig. 8-15, where

$$s^i \cdot d_j = C_j^i$$

The array of units and zeros of Fig. 8-15 is called a *logical basis* (see Sec. 10-5) for three diseases and three symptoms and is simply a listing of all conceivable sdc's.

We use the phrase "all *conceivable* sdc's"; but not all these are *possible*, or actually occur. It is *medical knowledge* that informs us which sdc's do or do not occur. For our example, suppose that medical knowledge consists of the following four observations:

Statement	*Symbolism*
1. A patient having $D(1)$ and also either $D(2)$ or $D(3)$ must have both symptoms $S(1)$ and $S(3)$	$D(1) \cdot [D(2) + D(3)] \rightarrow S(1) \cdot S(3)$
2. If a patient does not have $D(2)$, then he does not have $S(2)$	$\overline{D(2)} \rightarrow \overline{S(2)}$
3. If a patient does not have $D(1)$ but does have both $D(2)$ and $D(3)$, then he has symptom $S(3)$	$\overline{D(1)} \cdot D(2) \cdot D(3) \rightarrow S(3)$
4. A patient has at least one symptom and at least one disease†	$[S(1) + S(2) + S(3)] \cdot [D(1) + D(2) + D(3)]$

† In medical practice the presence of a complaint (symptom) must be regarded as presumptive evidence of disease, but the absence of a complaint cannot be equated with absence of disease.

The logical product of such statements is the Boolean function E that represents the relationships between disease and symptoms that comprise medical knowledge; i.e.,

$$E = E[S(1), \ldots, S(n), D(1), \ldots, D(m)]$$

For our example,

$$\begin{aligned} E = {} & \{D(1) \cdot [D(2) + D(3)] \rightarrow S(1) \cdot S(3)\} \\ & \cdot \{\overline{D(2)} \rightarrow \overline{S(2)}\} \cdot \{\overline{D(1)} \cdot D(2) \cdot D(3) \rightarrow S(3)\} \\ & \cdot \{[S(1) + S(2) + S(3)] \cdot [D(1) + D(2) + D(3)]\} \end{aligned}$$

The sdc columns crossed off in Fig. 8-15 represent sdc's that fail to satisfy one or more of these relations. For instance, consider the rightmost

crossed-off column, namely, $C_7{}^6$, which is

$S(1)$	0
$S(2)$	1
$S(3)$	1
$D(1)$	1
$D(2)$	1
$D(3)$	1

This sdc represents a patient having $D(1)$ and certainly $D(2)$ *or* $D(3)$, but not having $S(1)$, contrary to statement 1, and hence it has been crossed off. Another point of view is that the only columns of the basis that are to remain are those which each satisfy all the statements of medical knowledge. The remaining columns form what is called a *reduced* (constrained) basis (see Sec. 10-9).

Consider next a patient to be diagnosed. The patient presents a profile of certain symptoms that he does and does not have; however, in general there will remain other symptoms about which no information has been obtained from the patient. Such a possibly incomplete symptom complex is called the patient's *symptom profile* G. Similarly, a possibly incomplete disease complex relating to a patient we call the patient's *disease profile* f. The combination of these profiles is the patient's *symptom-disease profile*, $f \cdot G$. Thus the patient's profile G is a Boolean function of the symptoms:

$$G = G[S(1), \ldots, S(n)]$$

Now we are ready to state the logical problem: given the patient's symptom profile G and medical knowledge E, we desire to find the patient's disease profile f as a Boolean function of the diseases,

$$f = f[D(1), \ldots, D(m)]$$

For our example, suppose that the patient does not have symptom $S(1)$ but does have symptom $S(2)$; that is,

$$G = \overline{S(1)} \cdot S(2)$$

The only remaining sdc's that satisfy this condition are $C_2{}^2$, $C_2{}^6$, and $C_6{}^6$, corresponding to d_2 and d_6. Thus the diagnosis becomes

$$\begin{aligned} f &= \overline{D(1)} \cdot D(2) \cdot \overline{D(3)} + \overline{D(1)} \cdot D(2) \cdot D(3) \\ &= \overline{D(1)} \cdot D(2) \end{aligned}$$

Hence the patient does not have $D(1)$ and does have $D(2)$, but it is not known whether or not he has $D(3)$. As another example, let

$$G = S(1) \cdot S(2)$$

Then it is easily found that we have a diagnosis of four possibilities, namely, d_2, d_3, d_6, and d_7; thus

$$\begin{aligned} f = {} & \overline{D(1)} \cdot D(2) \cdot \overline{D(3)} + D(1) \cdot D(2) \cdot \overline{D(3)} \\ & + \overline{D(1)} \cdot D(2) \cdot D(3) + D(1) \cdot D(2) \cdot D(3) \\ = {} & D(2) \end{aligned}$$

which means that the patient has disease $D(2)$ and may or may not have $D(1)$ or $D(3)$. As a final example, let

$$G = S(1) \cdot S(2) \cdot \overline{S(3)}$$

i.e., let $S(3)$ be taken as a further test, and let the result be negative. Here

$$f = \overline{D(1)} \cdot D(2) \cdot \overline{D(3)}$$

which is a unique diagnosis which resulted from the additional test.

The methods of Chap. 10 can be utilized for determining the diagnostic possibilities, where the problem is one of solving the equation

$$G \rightarrow f$$

for the function f, *with respect to a basis constrained by the function* E.

Determining Further Tests. The ease with which medical symptoms or test results can be obtained varies greatly with the complexity of the examination, the cost, the inconvenience to the patient, and so forth. Thus, in any particular medical specialty, the examinations could be classified using difficulty of execution as the criterion. For example, it is often a fairly straightforward matter to obtain some aspects of a patient's medical history; then some parts of a physical examination are more convenient to perform than others; or some blood tests and X-ray examinations are less difficult and costly than others; and finally, certain tests, such as biopsies and exploratory operations, are used only when no other technique can provide the information needed. Clearly it is not feasible to make all possible tests on a patient before determining the diagnosis. The more realistic method uses iterated stages: making a group of tests, determining whether or not a precise diagnosis can be made from the results so far obtained, determining what other tests should be made, making these tests, etc. (see Fig. 8-16).

This problem can be formulated in terms of our logical considerations of the previous section. The formulation will be described by means of an illustration. Figure 8-17 shows a set of symptom-disease complexes associated with 14 symptoms and 8 diseases. For this illustration, the

symptoms have been classified into history, physical examination, laboratory blood tests, and bone marrow tests, in order of increasing difficulty of performance. Notice that $S(1)$ actually contributes nothing to the diagnosis: having a unit in every column, it does not distinguish between any columns. Suppose that the patient has both $S(2)$ and $S(3)$. Then the diagnosis so far is that he has a disease complex associated with one of the leftmost 11 columns. Because of the patient's symptoms, we can eliminate from consideration the rest of the columns shown shaded in Fig. 8-17: Of the physical-examination symptoms $S(4)$, $S(5)$, and $S(6)$, only two are necessary, for all three cannot better differentiate among the

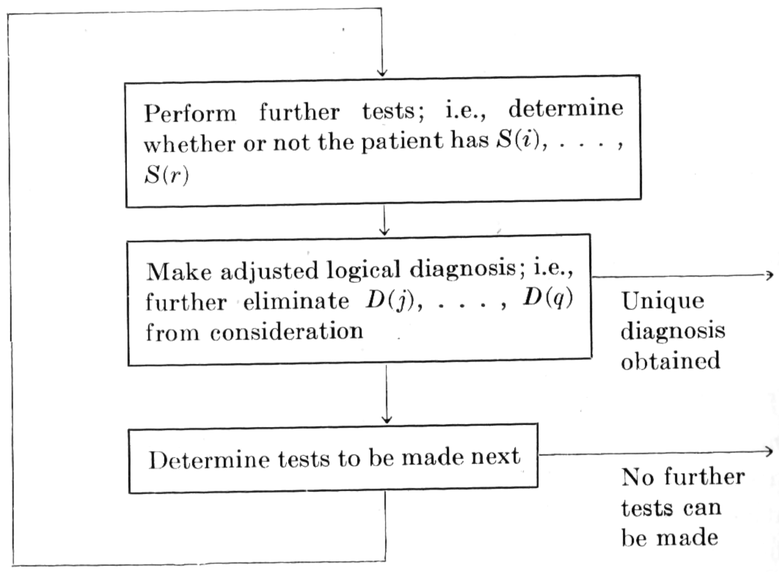


FIG. 8-16. The diagnostic-testing iteration.

remaining columns than can any two. Suppose that $S(4)$ and $S(5)$ are taken and that the patient has $S(4)$ and *not* $S(5)$; then only 7 columns remain as possible diagnoses, columns 3 through 9, as shown in Fig. 8-17. The choice among the blood tests $S(7)$ through $S(12)$ is not so easy to make. It can be shown that only three of these six tests need be made, namely, $S(9)$, $S(10)$, and $S(12)$. Suppose that the patient presents positive results for tests $S(9)$ and $S(12)$ and a negative result for test $S(10)$. Then only a single column remains, column 5, as shown in Fig. 8-17. Thus the bone marrow tests need not be made; the patient has a combination of diseases $D(4)$ and $D(5)$ and none of the other diseases. Suppose, on the other hand, that at the physical examination the patient had both $S(4)$ and $S(5)$. Then only columns 1 and 2 would remain, but the blood tests would not distinguish between them; bone marrow test $S(13)$ would have to be made.

Calculating the Alternative Diagnostic Probabilities. Frequently it is unfeasible or undesirable to make further tests to distinguish among alternative possible diagnoses. The problem then resolves into the question: In the light of the patient's present symptom profile G what is the probability that the patient has a particular disease complex f—that is, what is the *conditional probability* $P(f|G)$, where the term to the

Column numbers:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
History $S(1)$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
→ $S(2)$	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
→ $S(3)$	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0
Physical examination → $S(4)$	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0	0
→ $S(5)$	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0
$S(6)$	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1
Blood tests $S(7)$	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0
$S(8)$	1	1	1	1	1	1	1	0	1	1	1	0	0	0	0	1
→ $S(9)$	1	1	1	1	1	0	1	0	0	1	0	1	1	0	0	1
→ $S(10)$	1	1	1	1	0	1	0	0	0	0	0	0	0	1	0	0
$S(11)$	1	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0
→ $S(12)$	1	1	1	0	1	1	0	0	1	1	0	0	0	0	0	1
Bone-marrow tests $S(13)$	1	0	1	1	0	0	0	0	0	1	1	1	0	0	0	0
$S(14)$	1	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1
$D(1)$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$D(2)$	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0
$D(3)$	0	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
$D(4)$	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0
$D(5)$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
$D(6)$	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0
$D(7)$	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
$D(8)$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0

Symptom complexes

Disease complexes

FIG. 8-17. Logical basis for symptom-disease complexes associated with 14 symptoms and 8 diseases. After each set of diagnostic tests, the possible diagnoses are studied to determine what tests to do next.

right of the stroke is the condition (symptom profile) and the term to the left is the event (disease complex)? That probabilities can enter into a diagnosis depends on the fact that probability can enter into medical knowledge. For example, a medical textbook may read that, if the patient has disease j, then there is only a certain *chance* that he will have symptom i. At present it may generally be said that specific probabilities are rarely known; medical diagnostic textbooks rarely give numerical values. It is, however, a relatively simple matter to collect such sta-

tistics. Since we are considering topics from an essentially academic point of view, we shall assume that the probabilities are known and we shall discuss methods of utilizing such probabilities in medical diagnosis.

Consider, for example, a patient having symptom profile $\overline{S(1)} \cdot S(2)$, in the illustration of Fig. 8-15. The logical diagnosis results in two alternative diagnoses, d_2 and d_6. If we let

$$P(s^i \cdot d_j) = P(C_j{}^i)$$

then the probabilities for diagnoses d_2 and d_6 become

$$P[d_2|\overline{S(1)} \cdot S(2)] = \frac{P(C_2{}^2) + P(C_2{}^6)}{P(C_2{}^2) + P(C_2{}^6) + P(C_6{}^6)}$$

and

$$P[d_6|\overline{S(1)} \cdot S(2)] = \frac{P(C_6{}^6)}{P(C_2{}^2) + P(C_2{}^6) + P(C_6{}^6)}$$

We have illustrated the general probabilistic problem in medical diagnosis, which is to determine the conditional probabilities

$$P(d_j|G) \qquad \text{for each } j \in f$$

where by $j \in f$ we mean those values of j that index the alternative disease complexes remaining after the logical-diagnosis test analysis has been made and where G is the patient's symptom profile. This probability clearly depends on the characteristics of the population of patients involved and on the number of diseases under consideration. For proper application, it is important to observe that probability evaluations made at each stage of the logical-diagnosis test iteration will *not in any way* necessarily indicate a trend toward the final diagnosis. For example, consider a patient with a disease for which there is a specific test: until that test is made, the probability of the patient's having the disease might actually grow smaller and smaller with each iteration, but after the test is made, this probability will rise to unity! On the other hand, *when it becomes infeasible to make additional tests*, the probabilities must play an important role in the choice of treatment.

Bayes' Formula. Bayes' formula can give important insight into the types of ingredients which enter into a probabilistic diagnosis. For disease complex d_j and symptom complex s^i, the formula relates $P(d_j|s^i)$ with $P(s^i|d_j)$, as follows:

$$P(d_j|s^i) = \frac{P(d_j)P(s^i|d_j)}{\sum_{\text{all } k} P(d_k)P(s^i|d_k)}$$

where Σ over all k indicates summation over all the possible disease complexes under consideration. Let us focus our attention on the two terms

in the numerator, namely,

$$P(d_j)P(s^i|d_j)$$

since the denominator merely acts as a normalization factor. The term $P(s^i|d_j)$ is the probability of having the symptom complex s^i when it is known that the patient has the particular disease complex d_j. This is precisely what is described in medical textbooks when a particular disease is under consideration. Of course, textbooks do not at the present time assign numerical values to $P(s^i|d_j)$, but they do state that associated with a certain disease particular symptoms are common, frequent, rare, and so forth. The reason that $P(s^i|d_j)$ is discussed rather than $P(d_j|s^i)$ is that the etiology of the symptoms is related to, or stems from, the disease. Thus $P(s^i|d_j)$, that is, the symptoms resulting from a disease, *is generally a constant for the particular disease complex under consideration, being relatively independent of other circumstantial factors.*

On the other hand, consider the term $P(d_j)$. This is the probability that, of the particular population of patients involved, any particular patient has this disease complex d_i. By "particular population of patients," we mean that population from which the patient can be considered as chosen at random. For example, the population may be that of the entire United States, or that of a particular clinic in a particular hospital, etc.; in each case, $P(d_j)$ will in general be different. *This term relates the influence of geographical, seasonal, epidemiological, social, and other such factors to the diagnosis.* Thus, in the tropics one is more likely to find tropical disease; during an Asian flu epidemic, the symptoms of a cold are more likely to be related to Asian flu; some diseases, like poliomyelitis, have a seasonal variation; and so forth. A little reflection indicates clearly that both these factors—the circumstantial, or local, and the constant relating to the disease-symptom syndrome—influence the diagnosis. The denominator of Bayes' formula, $\sum_k P(d_k)P(s^i|d_k)$, being a normalizing factor, is rarely needed: this comes about because in evaluating the probabilities of alternative diagnosis it is the relative *ratios* of the probabilities that indicate the most likely diagnosis.

The significance of Bayes' formula now becomes clear. Since $P(s^i|d_j)$ is a constant, dependent only on the pathological-etiological aspects of the particular disease processes associated with d_j, tables of these conditional probabilities can be made. Then, for any particular population—such as a particular clinic, or the patients of a particular physician—the total probability $P(d_j)$ can be ascertained, as affected by the local environmental conditions, and the desired *diagnostic probability* $P(d_j|s^i)$ evaluated by the formula. This is, for example, what a physician qualitatively does when he tells a patient over the telephone: "Your symp-

toms of headache, mild fever, cough, general malaise, etc., indicate that you probably have Asian flu—many of my patients have it also—it's around our community, you know."

As noted above, we are actually interested in $P(d_j|G)$, where G is the patient's symptom profile—i.e., a Boolean function of the $S(i)$. In this case it is easy to see that Bayes' formula can be expressed as

$$P(d_j|G) = \frac{\sum_{i\in G} P(d_j)P(s^i|d_j)}{\sum_{i\in G}\sum_{\text{all } k} P(d_k)P(s^i|d_k)}$$

where $i \in G$ over Σ means that the summation is taken over all symptom complexes s^i included in G, that is, over all symptom complexes whose column positions correspond to the position of the units in the designation number of G. The analogous formula for $P(f|G)$, where f is a Boolean function of the diseases, becomes

$$P(f|G) = \frac{\sum_{j\in f}\sum_{i\in G} P(d_j)P(s^i|d_j)}{\sum_{i\in G}\sum_{\text{all } k} P(d_k)P(s^i|d_k)}$$

where the $j \in f$ under Σ indicates the summation over all disease complexes whose column positions correspond to the unit positions in the designation number of f.

If the symptoms can be considered *independent* (which can only be an approximation), then we can make use of the fact that in such a case

$$P[S(i) \cdot \cdots \cdot S(k)|d_j] = P[S(i)|d_j] \times \cdots \times P[S(k)|d_j]$$

and

$$P[\overline{S(i)}|d_j] = 1 - P[S(i)|d_j]$$

where it may be easier to determine $P[S(k)|d_j]$ than $P(s^i|d_j)$. For instance, in such a case we might have

$$P[d_6|S(1) \cdot S(2)] = \frac{P(d_6)P[S(1)|d_6]P[S(2)|d_6]}{\sum_{\text{all } k} P(d_k)P[S(1)|d_k]P[S(2)|d_k]}$$

and

$$P[d_2|\overline{S(1)} \cdot S(2)] = \frac{P(d_2)\{1 - P[S(1)|d_2]\}P[S(2)|d_2]}{\sum_{\text{all } k} P(d_k)\{1 - P[S(1)|d_k]\}P[S(2)|d_k]}$$

The Dynamic Diagnostic-treatment Problem. Suppose that alternative diagnoses remained for a patient after the logical analysis, that the probabilities of these diagnoses have been determined, and that treatment has been instituted. Clearly the effectiveness of the treatment on improving the patient's state of health will produce additional information about the alternative diagnoses and after a time may require a reevaluation of the choice of treatment. Such a treatment-diagnosis cycle, where each treatment itself acts like an additional test, is commonly utilized by physicians. However, it is the effect of the treatment on the patient's *state of health* that produces the required information.

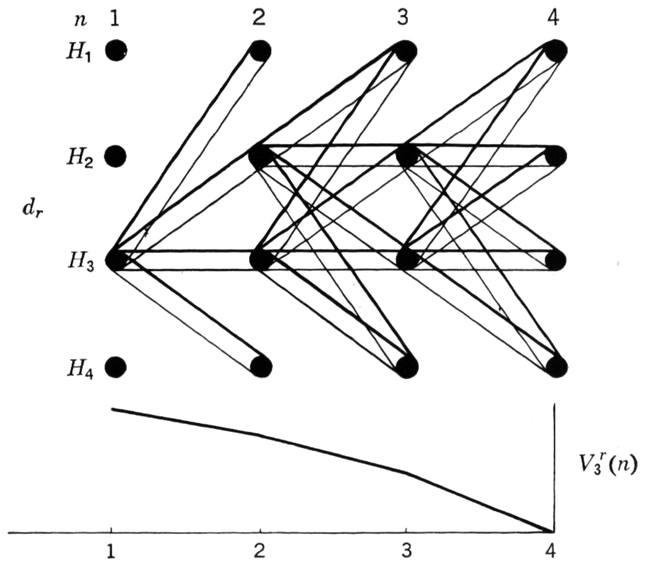


FIG. 8-18. Illustration of treatment-reevaluation plane for patient with disease complex d_r, starting in health state H_3. We assume (to simplify the drawing) that H_1 is the cured state and the patient will stay well and that H_4 is the critical state from which the patient cannot get well. Heavy lines represent transitions for treatment T_1, associated with p_{ij}^{r1} and v_{ij}^{r1}; light lines represent transitions for treatment T_2, associated with p_{ij}^{r2} and v_{ij}^{r2}. The graph indicates that $V_3{}^r(n)$ levels off in approximately four stages.

Suppose, for example, that four states of health of the patient can be distinguished: H_1, cured; H_2, good; H_3, poor; H_4, critical. Suppose also that there are two possible treatments T_1 and T_2 that can be given for each state of health. For each of the possible diseases d_r that the patient may have, let p_{ij}^{rk} and v_{ij}^{rk} be the transition probability and value, respectively, of going from state of health H_i to state H_j under treatment T_k. If we are concerned with only two disease complexes, say d_1 and d_2, then we would have a table similar to Table 8-6, column a, for each disease. For the moment, let us restrict ourselves to one of the diagnostic alternatives, say d_r, and consider the problem of choosing the "best" of the possible courses of treatments to "cure" this disease complex. Of course, after a treatment is instituted, the patient's health state may undergo a transition to another state, and the choice of treatment must be further

reevaluated, and so forth. The method for determining the choice of treatment that will maximize the expected value is dynamic programming, as discussed in the previous section. The only problem that remains is how to choose N, the number of iterations within which the

TABLE 8-7. VALUES FOR EXAMPLE IN TEXT

Health state	Treatment	Transition probabilities and values				Expectation	Boundary values
		$j = 1$	$j = 2$	$j = 3$	$j = 4$		
H_1							$V_1^r(N) = 1$
H_2	T_1 p_{2j}^{r1}	0.20	0.70	0.10	0	$E_2^1 = 0.22$	$V_2^r(N) = -1$
	T_1 v_{2j}^{r1}	0.30	0.20	−0.20	−10		
	T_2 p_{2j}^{r2}	0.40	0.30	0.30	0	$E_2^2 = 0.18$	
	T_2 v_{2j}^{r2}	0.50	0.30	−0.20	−10		
H_3	T_1 p_{3j}^{r1}	0.09	0.60	0.30	0.01	$E_3^1 = -0.78$	$V_3^r(N) = -3$
	T_1 v_{3j}^{r1}	0.40	0.30	0	−10		
	T_2 p_{3j}^{r2}	0.19	0.30	0.50	0.01	$E_3^2 = -0.69$	
	T_2 v_{3j}^{r2}	0.80	0.50	0	−10		
H_4							$V_4^r(N) = -10$

cure is to be effected. If the patient's health is initially in state H_i, the method is to graph $V_i(n)$ as the *backward* (!) calculation is accomplished (see Fig. 8-18); $V_i(n)$ will almost always rise to an asymptote—and when it gets close to the asymptote, the process may be discontinued and the number of backward steps will be a good value for N. Note

n	1	2	3	4	5	6	7	8	9	10	11	$12 = N$
H_2	T_1	T_1	T_1	T_1	T_1	T_1	T_2	T_2	T_2	T_2	T_2	T_2
H_1	T_1	T_1	T_2	T_2	T_2	T_2	T_2	T_2	T_2	T_2	T_2	T_1

FIG. 8-19. Optimum-treatment choice for the values of Table 8-7. The dashed lines denote places of change of treatment.

from Fig. 8-18 that we have assumed that when the patient is cured the probability of getting sick again is zero and that when the patient is in critical health the probability of getting well again is zero.

As an example of hypothetical probabilities and values, consider Table 8-7. Figure 8-19 gives the complete sequence of treatment decisions calculated from these values.

We are now ready to consider the case in which more than one diagnostic alternative remains. The diagnostic possibilities are mutually exclusive, and the patient must have one of them, say d_r, with probability P_r calculated by the methods discussed above. For each possible disease complex d_r, a calculation as described above is made, and for each d_r, an N_r, or number of cycles to plan for, is evaluated. The initial treatment is considered as being given at this N_r for each disease possibility,

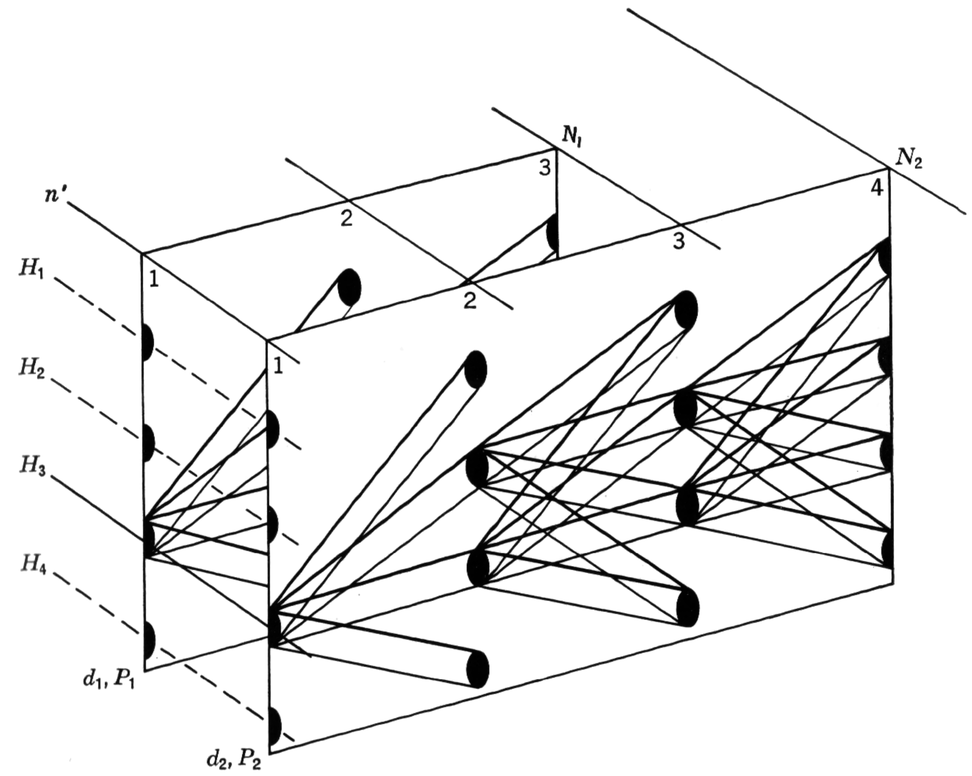


FIG. 8-20. Illustration of alignment of the d_1 and d_2 treatment-value planes for the start of the iteration.

as if the *treatment-value planes* of Fig. 8-18 for each disease d_r were aligned (see Fig. 8-20). Now we initiate the treatment-diagnosis iteration by computing for each k

$$E_i^k(n) = \sum_r P_r \left[E_i^k + \sum_{\text{all } j} p_{ij}^{rk} V_j^r(n) \right]$$

where for our example, to start with, $i = 3$. The initial treatment of choice becomes that T_k for which $E_i^k(n)$ is a maximum. After the treatment has been instituted for a sufficient period of time to evaluate its effectiveness, the transition in the patient's state of health, to H_j, is observed. This observation can now be utilized in the sense of another test result for the logical analysis. A reevaluation of the logical analysis

in the light of this new test result may reduce the number of alternative possible disease complexes that need be considered. Then the probabilities of the remaining disease complexes are reevaluated, resulting in a new collection of probabilities P_r. These probabilities are utilized in evaluating, for each treatment T_k,

$$E_i^k(n+1) = \sum_r P_r \left[E_i^k + \sum_{\text{all } j} p_{ij}^{rk} V_j^r(n+1) \right]$$

where now r takes on a restricted number of different values. The k corresponding to max $E_i^k(n+1)$ is taken to indicate the treatment of choice, and the iterations continue, as indicated in Fig. 8-21.

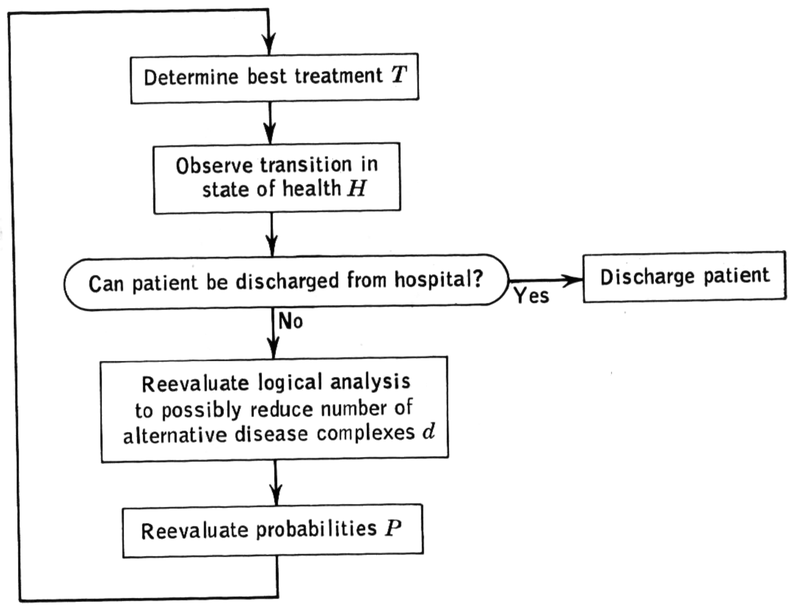


FIG. 8-21. Flow chart of the dynamic diagnostic-treatment cycle.

The iteration ends when a number of cycles $n = \min(N_r)$ has been accomplished or when the patient is cured, whichever occurs first. If the patient still has not been cured, then the process is reinitiated for the remaining disease complexes, just as described before.

8-6. Inductive Inference: Heuristic Programming

Meaning of Heuristic Programming. Inductive inference is the process of hypothesis making in a trial-and-error fashion, in order to draw conclusions or to solve problems. Whereas deductive inference is related to the mathematical methods currently in common use (as described in the two previous sections), *inductive inference* is related to the ingenuity and resourcefulness utilized in originally formulating such mathematical

methods. Inductive inference, then, is the problem-solving process as carried out by people, which can be described roughly as follows:†

> The work begins routinely, and then suddenly the person experiences a flash of understanding. This is followed by the writing down and checking of the solution. Apparently the person first used heuristic methods to look for a solution. To each suggestion turned up by the heuristic methods he applies some sort of test. The flash of understanding comes when a suggestion gets a high score on the test.

Heuristic‡ *programming* is the attempt to simulate on a computer the processes of inductive inference. It is the attempt to program computers to solve problems by somewhat the same methods that "intelligent" beings use. For example, when a man plays chess, he utilizes his intuition and experience to select only a certain small collection of all possible moves to be carefully examined—rather than determining the consequences of, say, all the possibilities two moves ahead (a little less than 1 million). A heuristic program utilizes such "intelligent," or skilled, guesses of what is best to examine in the attempt to solve a problem. Such a heuristic approach appears to be the most promising approach for the programming of computers in problems that are very loose-structured, extremely complex, and otherwise sufficiently difficult that man requires "intelligence" to solve them. For such problems, "good" hypotheses for trial are chosen—using as criteria for "goodness" certain available information that may suggest which possible solutions should be tried first, or that may rule out whole classes of solutions from being tried, or that may provide quick tests for distinguishing likely solutions from unlikely ones, and so on. Such criteria are called the *heuristics* of the problem. The field of heuristic programming is concerned with the investigation and understanding of various aspects of heuristics, such as how they are discovered, what kinds there are, what heuristics are used by humans, and so forth.

In the present section we shall illustrate the problems involved in heuristic programming by means of the example of proving plane-geometry theorems; on the basis of experience gained in this discussion, in the next section we shall describe an attempt at a "general problem-solver" program. It should be mentioned that plane geometry does yield

† Quoted from H. L. Gelernter and N. Rochester, Intelligent Behavior in Problem-solving Machines, *IBM J.*, vol. 2, no. 4, pp. 336–345, October, 1958. The present section is mainly based on this article.

‡ In standard usage, the term "heuristic" applies to discussions which stimulate in their auditors or readers the desire to investigate the subject further. Machines, however, must have this "desire" programmed in, of course, and the term has been extended to the process of investigation itself, to cover its empirical and probative aspects.

to a "decision algorithm"—i.e., a *deductive* technique for proving all theorems in plane geometry. However, the problem here is not to study plane geometry but rather to study how to program the computer to find proofs for generating (geometric) theorems, in the same manner that an intelligent high-school student would—using ingenuity, coupled with trial diagrams, etc., as a "heuristic" for suggesting possible "good" approaches.

Proving Geometric Theorems. To understand the meaning of a "proof" in geometry, we must first examine geometry from a mathematical point of view. To a mathematician, the theorems of geometry are *formal relations* among abstract entities, which basically need have no physical reality or significance—but which can be related to "points" and "lines" in special pictures if this seems interesting for the purpose at hand. To the computer, geometry must be considered as such a *formal system*, but the computer will use the interpretation of geometry as lines and points in space as a source for heuristics. A formal mathematical system has four ingredients:

1. A set of primitive symbols
2. Rules for the formation of well-formed formulas
3. Axioms
4. Theorems

The idea of primitive symbols and the concept of well-formed formulas are not new to us and have been used previously in the development of the ALGOL and COBOL automatic-programming languages (see Chaps. 6 and 7). For our geometry, the set of primitive symbols comprise the characters that represent *names* of points and lines (considered as undefined abstract entities) and the symbols that represent *relations* between the points and lines. The rules specify how to assemble the primitive symbols into well-formed formulas that are to have "meaning" in geometry, whether they be true or false. For instance, "Three points always lie on a straight line" is a well-formed formula (but false), whereas "Sides triangle of parallel every" is not well formed.

An axiom is a well-formed formula, such as "Two points can lie on one and only one straight line," which is regarded as being true by definition. From a set of axioms, new well-formed formulas can be *derived* as being true by means of "rules of inference," or *laws of logical reasoning.* Also, new well-formed formulas can similarly be derived (as being true) from previously established well-formed formulas. A *proof* is a sequence of well-formed formulas, in which each formula (or *line of proof*) either follows from preceding formulas by a rule of inference or else is itself an axiom or previously established theorem. A *theorem* is the last line in a proof.

Thus, in order for the computer to establish the proof of a theorem, it must produce a sequence of well-formed formulas whose *last formula* is the theorem and whose *first formula* is an axiom or previously established theorem and in which *every other formula* of the sequence is either derivable from preceding formulas or is an axiom or previously established theorem. It must be pointed out that the *finding* of such a proof, or sequence of well-formed formulas, is quite different from the *checking* of a given proof. For checking a proof, each successive line of the proof would be examined in relation to the previous lines, to determine which law of logic was used for its derivation. This should be a relatively straightforward problem to program for a computer: each formula, or line of proof, would be a list of characters or words in the computer's memory, and the laws of logic would be subroutines. We shall assume that such a proof-checking routine has already been written for our computer.

On the other hand, the process of *finding a proof* requires ingenuity and imagination, which are used by our intelligent high-school student. We might give our high-school student some helpful hints, such as "If you are trying to prove that two lines are parallel, try using the theorem 'If the opposite interior angles, formed by the intersection of two lines with a third, are equal, then the two lines are parallel.'" Drawing a picture might help suggest a method of approach to finding the proof. Such a figure might give insight into the problem. These are heuristic aids, and we shall now consider how a computer can be programmed to utilize such heuristic aids.

The Heuristic-programming Technique. In general, human beings do not solve problems directly but break them down into easier subproblems whose totality is equivalent to the original problem. This breaking down of a problem is a regression technique, which, of course, may not lead to a solution at all! In fact, this is one of the characteristics of the heuristic method—there is no guarantee that a solution will actually be obtained. To illustrate the point, consider the following argument:†

> I want to take my son to nursery school. What's the difference between what I have and what I want? One of distance. What changes distance? My automobile. My automobile won't work. What's needed to make it work? A new battery. What has new batteries? An auto repair shop. I want the repair shop to put in a new battery; but the shop doesn't know I need one. What is the difficulty? One of communication. What allows communication? A telephone. . . .

From the original goal, such an analysis forms subgoals, functions that need to be performed, and operations to perform them. For instance, let G_0 be the goal established by the problem of formulating a geometric

† From A. Newell, J. C. Shaw, and H. A. Simon, Report on a General Problem-solving Program, *Paper* P-1584, pp. 8–9, Rand Corp., Santa Monica, Calif., 1959.

proof—that is, G_0 is the formal statement or theorem to be proved. If G_1 is another formal statement from which G_0 can be inferred, then G_1 is called a subgoal. In general, if G_{i-1} can be inferred from G_i, then G_i is a subgoal of order i and all G_j for $j > i$ are called higher subgoals than G_i. The problem-solving graph of Fig. 8-22 summarizes the regression of

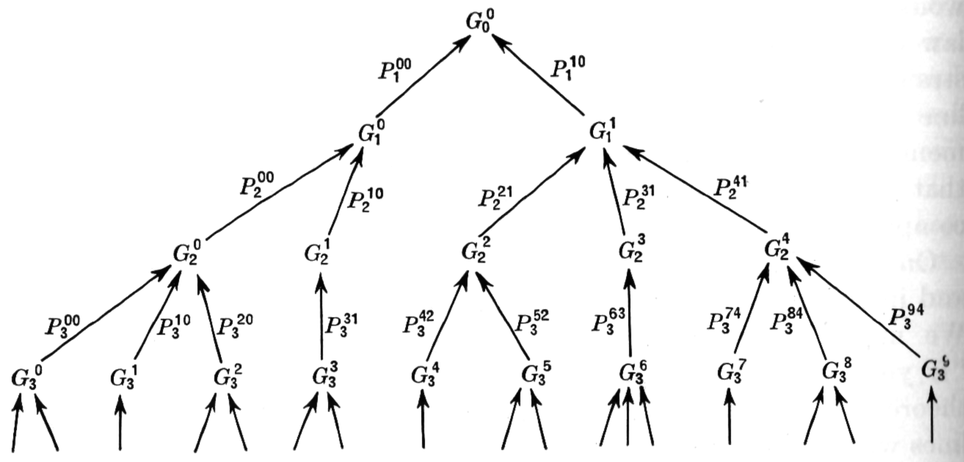


FIG. 8-22. Problem-solving graph illustrating a structure of subgoals G_i^{α} and transformation operators $P_1^{\alpha\beta}$.

subgoals; the G_i^{α} represent subgoals of order i, with α numbering the subgoals of a given order, and each $P_i^{\alpha\beta}$ is an operator that can transform a given G_i^{α} into G_{i-1}^{β}. The geometric problem is solved when any G_i can be obtained from the hypotheses and axioms.

The heuristics are utilized in developing the problem-solving graph during the heuristic processing. There are in general two types of heuristics, *syntactic heuristics* and *semantic heuristics*—named for the type of entity examined when the heuristic is being applied. We shall first illustrate the concept of a syntactic heuristic in geometry, as introduced by the following example. Suppose we are to prove the following theorem:

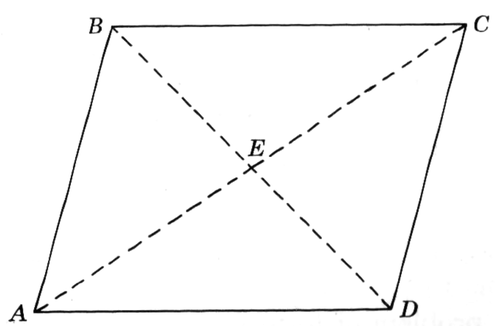


FIG. 8-23. Heuristic-aid figure for proving that the diagonals of a parallelogram bisect each other.

The diagonals of a parallelogram bisect each other.

From Fig. 8-23 we see that to prove the theorem we must establish the subgoals

$$G_0^0\colon AE = EC$$

and

$$G_0^1\colon BE = ED$$

The computer should be able to recognize, as people usually do, that the proof of the right-hand formula is essentially the same as that for the left-hand formula, and that therefore only one of these need be established. Similarly, we would not want the computer, after starting out to prove that $\Delta AEB \cong \Delta DEC$, to set up the subgoal of proving that

$$\Delta AED \cong \Delta BEC$$

These requirements are met if the syntactic heuristic of *symmetry* is utilized.

Let π be a permutation on the names of the syntactic variables in a theorem; then π is called a syntactic symmetry of the theorem if its operation on the hypotheses leaves them unchanged except for a possible transformation into an equivalent form with respect to the symmetries of the symbols. If $\{H\}$ is a set of hypotheses and π is a syntactical symmetry for a theorem, then

$$\pi\{H\} = \{H\}$$

We can now state in terms of π the important observation upon which the syntactic heuristic of symmetry is based: *If* Γ *is a well-formed formula provable from* $\{H\}$, *then* $\pi\Gamma$ *is a well-formed formula provable from* $\{H\}$. The proof of this observation is as follows:

If $\{H\} \rightarrow \Gamma$	is true
then $\pi\{H\} \rightarrow \pi\Gamma$	because renaming the syntactic variables in a theorem does not destroy the validity of the theorem
But $\pi\{H\} = \{H\}$	by hypothesis
so that $\{H\} \rightarrow \pi\Gamma$	is true, as desired

The application of our observation as a heuristic in aiding the selection of appropriate subgoals to a problem is as follows: G_i is not a suitable subgoal if it is syntactically symmetric with any other subgoal G_j for $j < i$ (since the same proof for G_j would hold for G_i with only a permutation of syntactical variables). For example, consider the problem of Fig. 8-23; the computer will eliminate G_0^1 from consideration since

$$\pi G_0^1 = G_0^0$$

where π transforms A into B, B into C, C into D, and D into A. Hence after proving G_0^0 the computer will merely assert "similarly G_0^1." Also, if at some point $\Delta AEB \cong \Delta DEC$ is a subgoal, the heuristic of symmetry will eliminate $\Delta AED \cong \Delta BEC$ as a possible subgoal, and so forth.

Such syntactical heuristics will be essential to the success of the geometric-theorem-proving program, to eliminate redundant paths and present an efficient problem-solving graph. However, such syntactical

heuristics do not tell which path the computer is to try on the graph; for this information, semantic heuristics are required. For example, one of the first heuristics that should be applied for proving geometric theorems with the aid of diagrams is the following:

If the diagram consists of a "bare" simple polygon, a construction will probably be required.

If a construction is indicated, a heuristic about which construction to make might be this:

If the figure has an axis of symmetry and it is not drawn, then draw it.

Applying such semantic heuristics to our problem, we would draw the lines AC and BD intersecting at E. Another useful heuristic will be this:

If the theorem asks that two line segments or angles be proved equal, determine by measuring whether these are corresponding parts of apparently congruent triangles. If so, attempt to prove the congruence.

Applying this heuristic we would develop the subgoal to prove that $\Delta AEB = \Delta DEC$. Another frequently used heuristic is this:

To prove two angles are equal, see whether they are interior opposite angles formed by a transversal of two parallel lines or whether they are opposite angles formed by two intersecting lines.

Applying this heuristic, we can set as subgoals that $\measuredangle BAE = \measuredangle ECD$ and $\measuredangle AEB = \measuredangle DEC$, which would show that $\Delta AEB \cong \Delta DEC$. Finally, to prove that $AE = EC$, the subgoal $AB = DC$ would be set up; but this follows from the hypothesis of the theorem and hence the proof is constructed.

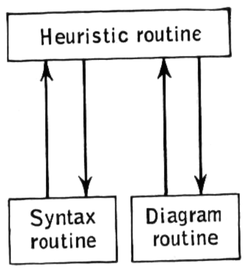


FIG. 8-24. Block diagram of the program to prove geometric theorems.

Programming Organization. The program for proving geometric theorems, constructed by H. L. Gelernter and N. Rochester, has three parts, a *syntax routine*, a *diagram* (or *semantic*) *routine*, and a *heuristic routine* (see Fig. 8-24). The syntax routine forms and keeps track of the lists of syntactical entities that comprise the lines of the proof and also checks the heuristically suggested trials. The diagram, or semantic, routine makes constructions and measures them (by means of analytic geometry); its output is *not* the values of the measurements but rather only qualitative information of the type acquired by a mathematician scanning a well-drawn figure. The heuristic routine is the boss, so to speak; it decides what to do next, directing the diagram routine to perform certain constructions and the syntax routine to test lines of proof, and so forth.

8-7. Inductive Inference: The General Problem-solving Program

Concept of a General Problem-solving Program. The fact that a heuristic program basically operates on syntactical collections, or formulas, to obtain new formulas (e.g., operator $P_i^{\alpha\beta}$ transforms G_i^{α} into G_{i-1}^{β}) reminds one of our translator of Sec. 8-2, which also operated on syntactical entities. Of course, the translator involved *no* heuristic considerations. But it did embody those concepts that related to programming-language translation in general, and operated upon those entities that related to the source or target language in particular; and hence it could be applied to any source-to-target-language translation. The question immediately arises of whether, in a similar manner, the general aspects of a heuristic program could be separated from the particular and a *general problem-solving program* constructed that would handle many different heuristic-programming problems when provided with the particular information required for a particular problem. A. Newell, J. C. Shaw, and H. A. Simon have indeed suggested such a general problem solver, which is the topic of this section.†

The general problem-solving program is based on the results of an analysis of heuristics in which three general *heuristic types* are distinguished. Each such type of heuristic is associated with a goal-related *transformation type*. We now describe these transformation heuristic-type combinations (see Fig. 8-25). The first goal-related transformation type is concerned with transforming the subgoal form G_i under consideration at any stage of the problem into the desired final goal form G_0. For example, in the process of demonstrating the trigonometric identity

$$(\tan \theta + \cot \theta) \sin \theta \cos \theta = 1$$

a subgoal might be to transform

$$G_i\text{: } \left(\frac{1}{\cot \theta} + \cot \theta\right) \sin \theta \cos \theta$$

into

$$G_0\text{: } 1$$

The heuristic for this first type is *to determine*, if possible, *the difference between G_i and G_0* and then *reduce this difference* (see Fig. 8-25, heuristic *a*). The difference between a G_i and G_0 could be, for instance, differences in the variables that appear in each (called ΔV differences) or differences in the number of times a variable occurs in each (called ΔN differences), and so forth. For our illustration both ΔV and ΔN differences exist.

† See *ibid.* The techniques and examples of this section are based on that report.

Reducing the difference between goal forms is a goal-related transformation of the second type, and leads to the heuristic of the second type (see Fig. 8-25, heuristic *b*). The difference is reduced by *searching for an operator* Q that can reduce the kind of difference in question and then *applying the operator to* G_i. For example, consider the operator that

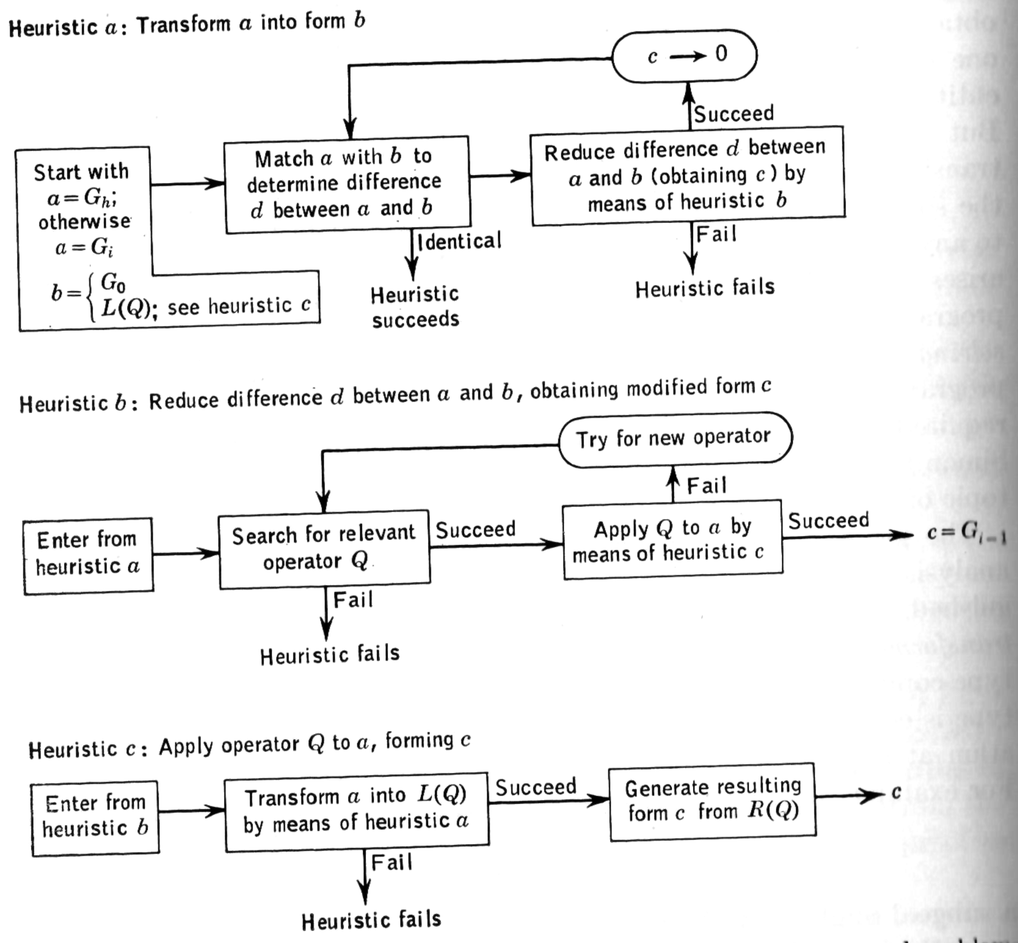


FIG. 8-25. Block diagrams of the three types of heuristics of the general problem-solving program: G_h is the hypothesis, or starting form; G_o is the desired final form; and G_i is a subgoal form.

turns cot x into cos x/sin x; when applied to our illustrative G_i, the result becomes

$$\left(\frac{1}{\cos\theta/\sin\theta} + \frac{\cos\theta}{\sin\theta}\right)\sin\theta\cos\theta$$

which has only two different variable forms (sin θ and cos θ) instead of the three (sin θ, cos θ, and cot θ) of the original G_i. Hence the difference ΔV between G_i and G_0 has been reduced by the application of the operator.

Actually applying the operator to G_i requires a heuristic of the third type (see Fig. 8-25, heuristic *c*). An operator Q has two sides, one at

the tail of the arrow and one at the tip of the arrow, denoted by $L(Q)$ and $R(Q)$, respectively. For instance, if Q is the operator that turns cot x into cos x/sin x, it is written as

$$\cot x \rightarrow \frac{\cos x}{\sin x}$$

where $L(Q) = \cot x$ and $R(Q) = \cos x/\sin x$. A more complicated operator might be

$$\sin^2 x + \cos^2 x \rightarrow 1$$

In some cases, it might not be possible to apply Q, because G_i may not be in the appropriate form. In such a case, a goal-related transformation of the first type is required, except that now the difference between G_i and $L(Q)$ is to be reduced. In any event, a substitution of variables is usually required (for example, x for θ), and hence when $R(Q)$ is obtained, a replacement of variables results in G_{i-1}.

In order to apply the three types of heuristics of the general problem-solving program to a particular problem, certain particular information must be given concerning the nature of the structures with which the problem is concerned. This is analogous to the particular syntactical and semantic description of the object language in terms of the target language required in order to apply the translator program in a particular case. For the general problem-solving program, the following information must be supplied: (1) the syntactical description of well-formed formulas; (2) a list of the available operators [of the form $L(Q) \rightarrow R(Q)$]; (3) definitions of the kinds of differences that may exist; and (4) a table of the connections between these differences and the operators capable of producing them [although such a tabulation could be derived from (2) and (3)]. In addition, the heuristics must be applied in such a way that progress toward the goal is made; hence we must also be given (5) criteria for evaluating relative progress toward the goal among different difference reductions. We shall now present an example of the use of the program to prove trigonometric identities in the manner of an "intelligent" high-school student.

Example of Proving Trigonometric Identities. As an example of the method, consider a problem in trigonometry. First, we must state the required information concerning trigonometry, which is as follows:

1. SYNTACTICAL ENTITIES. Ordinary algebraic expressions, including the trigonometric functions. The associative and commutative laws are considered as being implicit in the notation: the program can select freely which terms to use in an expression like $x + y + z$. The syntactical definitions are similar to those given above (in Sec. 8-3) for the simple arithmetic expression but also include the trigonometric functions.

2. OPERATORS (a double arrow means the transformation works both ways)

Name	*Definition*
*A*0	These are recursively defined to be applied from the innermost subexpressions to the main expression: $A + (B + C) \rightarrow A + B + C$; $A(BC) \rightarrow ABC$; $A + 0 \rightarrow A$; $A + A \rightarrow 2A$; $A - A \rightarrow 0$; $0A \rightarrow 0$; $1A \rightarrow A$; $AA \rightarrow A^2$; $A^B A^C \rightarrow A^{B+C}$; $A^0 \rightarrow 1$; $0^A \rightarrow 0$; $A^1 \leftrightarrow A$; $(A^B)^C \rightarrow A^{BC}$; $(AB)^C \rightarrow A^C B^C$
*A*1	$(A - B)(A + B) \leftrightarrow A^2 - B^2$
*A*2	$(B + C)A \leftrightarrow BA + CA$
*A*3	$(A + B)^2 \leftrightarrow A^2 + 2AB + B^2$
*T*1	$\tan x \leftrightarrow \dfrac{1}{\cot x}$
*T*2	$(\tan x)(\cot x) \leftrightarrow 1$
*T*3	$\tan x \leftrightarrow \dfrac{\sin x}{\cos x}$
*T*4	$\cot x \leftrightarrow \dfrac{\cos x}{\sin x}$
*T*5	$\sin^2 x + \cos^2 x \leftrightarrow 1$

3. DIFFERENCES (trigonometric functions are detected by ΔV and ΔN)

Name	*Description*
ΔV	A variable appears in one expression that does not appear in the other. For instance, $\sin^2 x$ is said to differ from $\sin x \tan x$ by $+V$, since it needs a $\tan x$; $A + \cos x$ is said to differ from A by $-V$ since it needs to lose the $\cos x$.
ΔC	There is a difference in the binary connective. For instance, $\sin x + \tan x$ differs from $\sin x \tan x$ by ΔC.
ΔN	A variable occurs different numbers of times in the two expressions. For instance, $\sin x + \cos x$ is said to differ from $\sin^2 x + \cos x$ by $+N$, since it needs another $\sin x$; $\sin^2 x$ is said to differ from $\sin x$ by $-N$, since it needs to reduce the number of occurrences of $\sin x$.
ΔT	There is a difference in the sign (+ or −) of the two expressions.

4. TABLE OF CONNECTIONS BETWEEN DIFFERENCES AND OPERATORS

	*A*0	*A*1 ⇌	*A*2 ⇌	*A*3 ⇌	*T*1 ⇌	*T*2 ⇌	*T*3 ⇌	*T*4 ⇌	*T*5 ⇌
ΔV	−				*t*	*t*	*t*	*t*	*t*
ΔC	✓	✓	✓	✓					
ΔN	−	∓	±	±	*t*	*t*	*t*	*t*	*t*
ΔT	✓								

The +, −, or ✓ in the table means that the operator of the column affects the difference of the row, as specified; for double-arrowed oper-

ators the differences specified correspond to the respective directions indicated under the operator name. The *t* means that the operator should be accepted for a "−" difference if $R(Q)$ contains functions which already occur in the expression (or subexpression) and also that it should be accepted for a "+" difference if $R(Q)$ contains functions which do not occur in the expression (or subexpression).

5. CRITERIA OF PROGRESS. All differences in subexpressions are less important than differences in expressions. For a pair of expressions the differences are ranked thus, from most to least important: $+V$, $-V$, ΔC, $+N$, $-N$, ΔT. For instance, in $\sin x \cos x$ versus $\sin^2 x + \cos x$, ΔC is more important than $+N$; and in $\sin x (\cos x + \cot x)$ versus $\sin x + (\cos x + \sin x)$, $-V$ is more important than ΔC.

Having presented the required information, we are now ready to tackle an example. Consider the proof of

$$(\tan \theta + \cot \theta) \sin \theta \cos \theta = 1$$

We shall utilize the block diagrams of Fig. 8-25, starting with heuristic *a*, with G_h (which is essentially the hypothesis) being

$$(\tan \theta + \cot \theta) \sin \theta \cos \theta$$

and G_0 being 1. The work follows; the student should follow in detail the utilization of Fig. 8-25 as the steps progress. Each step in the heuristic process is labeled in order with an $H\alpha$; the subgoals are labeled by G_{h-i} as they are generated. The indention of the lines follows the recursive use of the block diagrams of Fig. 8-25 for the heuristic types:

Given G_h: $(\tan \theta + \cot \theta) \sin \theta \cos \theta$
Final goal G_0: 1

*H*1. (Try to) transform G_h into G_0: match produces difference $-V$ $(\tan \theta)$.
*H*2. Reduce $-V$ between G_h and G_0: first operator found is *A*0.
*H*3. Apply *A*0 to G_h: no change produced; so find next operator *T*1.
*H*4. Apply *T*1 to G_h.
*H*5. Transform G_h into $L(T1)$: succeeds for subexpression $\tan \theta$, producing

G_{h-1}: $(1/\cot \theta + \cot \theta) \sin \theta \cos \theta$

*H*6. Transform G_{h-1} into G_0: match produces difference $-V$ $(\cot \theta)$.
*H*7. Reduce $-V$ between G_{h-1} and G_0: first operator found is *A*0.
*H*8. Apply *A*0 to G_{h-1}: no change produced; so find next operator *T*4.
*H*9. Apply *T*4 to G_{h-1}.
*H*10. Transform G_{h-1} into $L(T4)$: succeeds, producing

G_{h-2}: $[1/(\cos \theta/\sin \theta) + \cos \theta/\sin \theta] \sin \theta \cos \theta$

$H11$. Transform G_{h-2} into G_0: match produces difference $-V$ (cos θ).

$H12$. Reduce $-V$ between G_{h-2} and G_0: first operator found is $A0$.

$H13$. Apply $A0$ to G_{h-2}: transform† $1/(\cos\theta/\sin\theta)$ into $\sin\theta/\cos\theta$, producing

G_{h-3}: $(\sin\theta/\cos\theta + \cos\theta/\sin\theta)\sin\theta\cos\theta$

$H14$. Transform G_{h-3} into G_0: match produces difference $-V$ (cos θ).

$H15$. Reduce $-V$ between G_{h-3} and G_0: first operator found is $A0$.

$H16$. Apply $A0$ to G_{h-3}: no change produced; find next operator $T5$.

$H17$. Apply $T5$ to G_{h-3}.

$H18$. Transform G_{h-3} into $L(T5)$: match produces connective difference ΔC ($\cdot$ versus $+$).

$H19$. Reduce ΔC between G_{h-3} and $L(T5)$: first operator found is $A0$.

$H20$. Apply $A0$ to G_{h-3}: no change produced; find next operator $A1$.

$H21$. Apply $A1$ to G_{h-3}.

$H22$. Transform G_{h-3} into $L(A1)$: match produces difference ΔC.

$H23$. Reduce ΔC between G_{h-3} and $L(A1)$: *reject* because difference is no easier than difference in $H19$; that is, no progress is being made according to the criteria. Find next operator $A2$.

$H24$. Apply $A2$ to G_{h-3}.

$H25$. Transform G_{h-3} into $L(A2)$: match succeeds with $A = \sin\theta\cos\theta$, $B = \sin\theta/\cos\theta$, and $C = \cos\theta/\sin\theta$, producing

G_{h-4}: $(\sin\theta/\cos\theta)\sin\theta\cos\theta + (\cos\theta/\sin\theta)\sin\theta\cos\theta$

$H26$. Transform G_{h-4} into $L(T5)$: match produces difference ΔC in left-hand expressions.

† $A0$ can be used, since

$$\frac{1}{\cos\theta/\sin\theta} = [(\cos\theta)(\sin\theta)^{-1}]^{-1} = (\cos\theta)^{-1}[(\sin\theta)^{-1}]^{-1} = (\cos\theta)^{-1}\sin\theta$$

$H27$. Reduce ΔC between left expression of G_{h-4} and left expression of $L(T5)$.

$H28$. Apply $A0$ to left expression of G_{h-4}, obtaining†

G_{h-5}: $\sin^2\theta + (\cos\theta/\sin\theta)\sin\theta\cos\theta$

$H29$. Transform G_{h-5} into $L(T5)$: difference ΔC exists in right-hand expressions.

$H30$. Reduce ΔC between right expression of G_{h-5} and right expression of $L(T5)$.

$H31$. Apply $A0$ to right expression of G_{h-5}: succeeds, obtaining

G_{h-6}: $\sin^2\theta + \cos^2\theta$

$H32$. Transform G_{h-6} into $L(T5)$: succeeds, resulting in

G_{h-7}: 1

$H33$. Transform G_{h-7} into G_0: identical, and proof is concluded.

Thus the lines of the proof become

$$G_7: (\tan\theta + \cot\theta)\sin\theta\cos\theta$$
$$G_6: \left(\frac{1}{\cot\theta} + \cot\theta\right)\sin\theta\cos\theta$$
$$G_5: \frac{1}{\cos\theta/\sin\theta} + \frac{\cos\theta}{\sin\theta}\sin\theta\cos\theta$$
$$G_4: \left(\frac{\sin\theta}{\cos\theta} + \frac{\cos\theta}{\sin\theta}\right)\sin\theta\cos\theta$$
$$G_3: \frac{\sin\theta}{\cos\theta}\sin\theta\cos\theta + \frac{\cos\theta}{\sin\theta}\sin\theta\cos\theta$$
$$G_2: \sin^2\theta + \frac{\cos\theta}{\sin\theta}\sin\theta\cos\theta$$
$$G_1: \sin^2\theta + \cos^2\theta$$
$$G_0: 1$$

where we have let $h = 7$ in G_{h-i}.

† Using $A0$, we find

$$\frac{\sin\theta}{\cos\theta}\sin\theta\cos\theta = \sin\theta\sin\theta(\cos\theta)^{-1}\cos\theta = \sin^2\theta(\cos\theta)^{-1}(\cos\theta)^1$$
$$= \sin^2\theta(\cos\theta)^0 = (\sin^2\theta)(1) = \sin^2\theta$$

The Heuristic Method. Many additional features can be incorporated into the heuristic method. The concept of *planning*, or constructing a solution in general terms before working out the details, can be incorporated into the method. Planning can be accomplished by (1) generalizing the forms by omitting certain details of the original variables and connectives, (2) formulating the corresponding problem in terms of the generalized forms, (3) solving the generalized problem, where this solution is to be considered as the plan for solving the original problem, and (4) translating the plan back into the nongeneralized forms and utilizing it for solving the original problem. The details are beyond the scope of the present treatment. Another important concept that can be included in the heuristic method is that of *learning*, for certainly a human problem solver becomes more skilled at solving a particular class of problems as he gains more experience in practice. Learning can be accomplished by enabling past history (1) to improve the set of differences that are utilized and (2) to improve the process of selecting the operators utilized to remove the differences, by adjusting the table of connections. These and many other additional features of the heuristic method are the subject of a great deal of current research.

In conclusion, we might note how our use of the term "inductive inference," as illustrated in this and the previous section, falls short of other, more general connotations of the word.† For we could include under "inductive inference" the discovery of regularities of an environment, or the formulation of the laws that govern an environment, or the prediction of the likely consequences of an action in an environment, where the process of "discovery," "formulation," or "prediction" would depend on the computer's inductive capabilities as it "lived in" and reacted with the environment in question.

8-8. Abstraction and Creativity

Introduction. In this section, we shall avoid as far as is possible discussions of philosophy, epistemology, aesthetics, and other such dialectic topics, although we must admit that these topics are closely related to our subject of computer programming to achieve the "intelligent" capabilities of abstraction and creativity. Our purpose is merely to introduce briefly some of the methods of approach that could be taken in using the computer for attempting to achieve artificial intelligence in these areas. Consideration of abstraction falls naturally into two categories: The first is associated with the problem of *identifying* or *recognizing* a concrete entity as belonging to, or being an element of, a particular abstract or

† See, for example, M. Minsky, Steps toward Artificial Intelligence, *Proc. IRE*, vol. 49, no. 1, p. 27, January, 1961.

conceptual class. For example, how does one recognize a certain animal as a "dog," or a combination of symbols as a "simple arithmetic expression," or a pictorial figure as the letter "A," and so forth? The second is concerned with the problem of originally *formulating* the abstraction or concept from a plethora of experiences. For instance, how does one decide in the first place that animals can be classed† as "dogs," "cows," etc., or that there is an entity "simple arithmetic expression" which can be usefully distinguished from other mathematical forms, or that "rectangles," "circles," etc., are to be considered as distinguishable concepts, and so forth?

Programming to achieve creativity, as in music, graphic art, or mathematics, is closely related to the process of abstraction and will be introduced through the example of musical composition. In this section, however, all our discussions will be found inadequate and certainly incomplete; but as yet the analysis of the problems involved in programming computers to achieve abstraction and creativity has been incomplete and is the topic of vigorous current research. If the ideas presented in this section stimulate further discussion, if they raise more questions than they answer, if they create an awareness of the potentialities of computer capabilities, then we shall have fulfilled our purpose.

Deductive Methods in Abstraction. Consider the first problem in abstraction: that of identifying an object as a concrete representation of an already understood concept, sometimes referred to as *pattern recognition.* Such an identification can be made through either deductive or inductive inference, as described in the previous sections. For example, deductive inference in the form of the logical analysis of Sec. 8-5 (used there to identify a disease entity) may be interpreted as a general method for identifying an object as belonging to a class according to the configuration of its characteristics. To carry this interpretation further, the result of such attempted identification may involve only a probabilistic outcome, in which the object may be considered as a concrete representation of more than one concept. In such a case, further information concerning the object would be gathered and another attempt at identification would be made, continuing in such a recursive manner until a unique recognition is accomplished.

Alternatively, identification by means of deductive inference can involve a syntactical-type analysis. For example, the simple arithmetic expressions included in a collection of arithmetic symbols can be recognized through the application to the collection of the translator routine (Figs. 8-4 and 8-5) in conjunction with the syntax of Table 8-4. In the syntax given there, the only *syntactical relation* required was juxta-

† Nontaxonomically, that is!

TABLE 8-8. SYNTACTICAL RELATIONS FOR PICTORIAL EXPRESSIONS

Symbol	Meaning	Example	Representing
↑	is above	○ ↑ □	○ / □
→	is next to	○ → □	○ □
⊙	is inside of	○ ⊙ □	◎
◯	is inside on the bottom of	○ ◯ □	◎
↥	rests on top of	△ ↥ □	⌂
↦	rests next to	□ ↦ □	□□

position, indicated by adjacency, e.g., ⟨letter⟩⟨letter⟩ for AB. Additional syntactical relations can enable the syntactical definitions to be formed of, for example, pictorial figures. As an illustration, consider the relations of Table 8-8. In terms of such relations we can syntactically define a four-year-old boy's concept of a "house" as follows:

⟨door⟩ : : = ▯

⟨windows⟩ : : = ⊞ | ⟨windows⟩ → ⊞

⟨chimney⟩ : : = ▫ | ▯

⟨wall⟩ : : = □ | ⟨door⟩ ◯ □ | ⟨windows⟩ ⊙ □

⟨gable⟩ : : = △ | ⟨chimney⟩ ↥ △

⟨roof⟩ : : = ▱ | ⟨chimney⟩ ↥ ▱

⟨front view⟩ : : = ⟨gable⟩ ↥ ⟨wall⟩

⟨side view⟩ : : = ⟨roof⟩ ↥ ⟨wall⟩

⟨house⟩ : : = ⟨front view⟩ | ⟨house⟩⟨side view⟩

Such a description will recognize the pictures of Fig. 8-26*a* as houses but will not so identify the pictures of Fig. 8-26*b*. With this notation, each of the pictures of Fig. 8-26*a* can be written in terms of a *pictorial expression*, respectively as follows:

△ ↥ □

(▯↥△) ↥ (▯◯□)

[△↥(▯◯□)] ↦ [(▯↥▱)↥(⊞⊙□)]

[△↥(▯◯□)] ↦ {[(▯↥▱)↥(⊞⊙□)] ↦ [▱↥(⊞ → ⊞) ⊙ □]}

Inductive Methods in Abstraction. Concept identification, or pattern recognition, can also be accomplished through inductive inference by means of heuristic programming, which has been described in the previous sections. The problem then becomes one of eliminating the "difference"

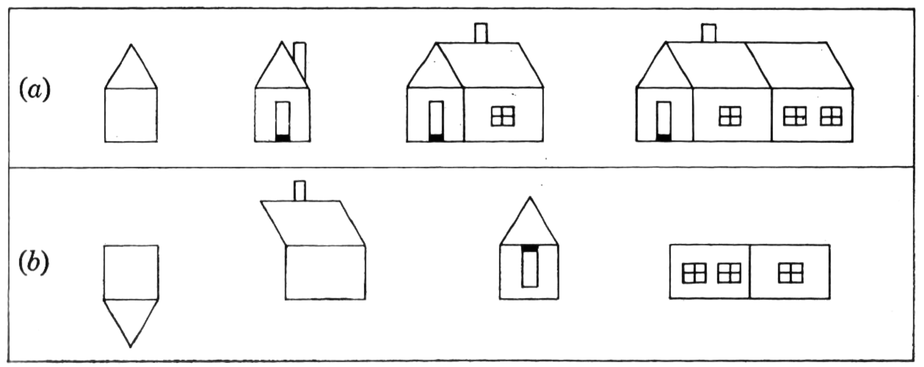


FIG. 8-26. (*a*) Pictures recognized as houses; (*b*) pictures not identified as houses.

between the given object and the *idealized concept* of the abstraction; if the difference can be completely reduced, the identification will have been made. In order to give an example, we must first define the five parts of the subject matter upon which the heuristic methods of Fig. 8-25 are to work. Suppose these are as follows:

1. OBJECTS. Pictorial expressions as defined above.
2. OPERATORS

*P*1 ⊞ → ⊞ ⟹ ⊞

*P*2 ▯ ◯ □ ⟹ □

*P*3 ⊞ ⊙ □ ⟹ □

*P*4 ▫↥▱ ⟹ ▱

*P*5 ▱ ↥ □ ⟹ □

*P*6 ▯↥△ ⟹ △

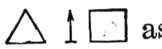
*P*7 (△↥□) → □ ⟹ △↥□

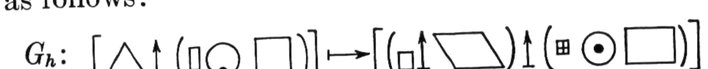
3. DIFFERENCES. Only ΔV and ΔN occur, and are defined as in Sec. 8-7.

4. CONNECTIONS. t means accept if $R(Q)$ contains figures that also occur in the pictorial expression:

	$P1$	$P2$	$P3$	$P4$	$P5$	$P6$	$P7$
ΔV		t	t	t	t	t	
ΔN	t						t

5. CRITERIA OF PROGRESS. Defined as in Sec. 8-7 (for remaining difference).

To identify a pictorial expression as a "house," we shall choose  as G_0, the idealized concept house, and try to reduce to nothing the difference between the given expression G_h and this G_0: if we are successful, the given expression is a house; if not successful, it is not a house. (The "ideal" house is the left diagram in Fig. 8-26*a*.) Consider as an illustration the expression  , which is the second diagram from the right in Fig. 8-26*a*. The work proceeds as follows:

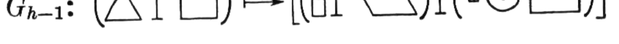
G_h: 

G_0: 

*H*1. Transform G_h into G_0.

*H*2. Reduce ΔV between G_h and G_0 ( ): first operator is $P2$.

*H*3. Apply $P2$ to G_h, finding

G_{h-1}: 

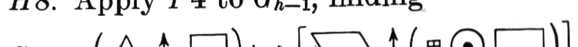
*H*4. Transform G_{h-1} into G_0.

*H*5. Reduce ΔV between G_{h-1} and G_0 (): first operator is $P2$.

*H*6. Apply $P2$ to G_{h-1}: no change produced; next operator is $P3$.

*H*7. Apply $P3$ to G_{h-1}: no change produced; next operator is $P4$.

*H*8. Apply $P4$ to G_{h-1}, finding

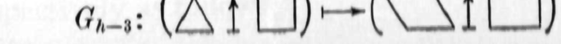
G_{h-2}: 

*H*9. Transform G_{h-2} into G_0.

*H*10. Reduce ΔV between G_{h-2} and G_0 (): first operator is $P2$.

*H*11. Apply $P2$ to G_{h-2}: no change produced; next operator is $P3$.

*H*12. Apply $P3$ to G_{h-2}, finding

G_{h-3}: 

*H*13. Transform G_{h-3} into G_0.

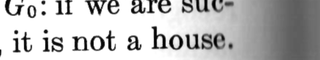
*H*14. Reduce ΔV between G_{h-3} and G_0: first operator is $P2$.

*H*15. Apply $P2$ to G_{h-3}: no change produced; try $P3$.

*H*16. Apply $P3$ to G_{h-3}: no change produced; try $P4$.

*H*17. Apply $P4$ to G_{h-3}: no change produced; try $P5$.

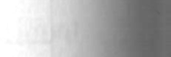
*H*18. Apply $P5$ to G_{h-3}, finding

G_{h-4}: 

*H*19. Transform G_{h-4} into G_0.

*H*20. Reduce ΔN between G_{h-4} and G_0 (): first operator $P7$.

*H*21. Apply $P7$ to G_{h-4}, finding

G_{h-5}: 

*H*22. Transform G_{h-5} into G_0: identical, and hence G_h is identified as a house.

One of the most interesting aspects of this formulation of the identification of abstractions, or pattern recognition, is the speculation that this is the method most closely related to brain mechanisms. For if one asks, say, a four-year-old child to draw a house, he will draw his *idealized conception of a house;* this in all probability will actually look like the left-hand diagram in Fig. 8-26*a*. Thus, according to this theory, the child recognizes a house by reducing it to his conceptual idealization. Such a speculation will explain why a child, when asked to draw a table, will almost always display *four* legs and a flat top, completely out of perspective; this is the idealized conception of a table as syntactically recorded in the child's brain! Further discussion of these fascinating ideas is, unfortunately, beyond the scope of the present treatment.

Formulation of Abstractions. The second aspect of abstraction is associated with the original formulation of the abstraction, or concept, itself. The above study of the process of identifying an object as a representation of an abstraction, or concept, can shed much light upon this formulation process. Again we can distinguish between concept formulation through the utilization of deductive inference and concept formulation through the use of inductive inference. In the deductive-inference case, for example, we could list the characteristics of a representative sample of the population under consideration, for instance as in Table 8-9. For concreteness, suppose the columns represent medical

TABLE 8-9. REPRESENTATIVE SAMPLE OF THE POPULATION

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}
$S(1)$	1	1	1	1	1	0	0	0	0	0	0	0
$S(2)$	0	0	0	0	0	1	1	1	1	1	1	1
$S(3)$	0	0	0	0	0	0	1	0	0	0	1	0
$S(4)$	1	1	1	0	1	0	0	0	0	1	1	1
$S(5)$	0	1	0	0	1	1	1	1	1	0	0	0
$S(6)$	0	0	1	0	0	0	0	1	0	0	0	0
$S(7)$	0	0	0	1	0	0	0	0	0	0	0	1
$S(8)$	0	0	0	0	0	1	1	1	1	0	0	0
$S(9)$	0	0	0	0	0	0	0	0	1	1	1	1
$S(10)$	0	0	0	0	1	0	0	0	0	1	1	1

patients and the rows symptoms, as in Sec. 8-5, a 1 indicating that the symptoms are present, a 0 that it is absent. In order to determine the different diseases that may be involved, i.e., to classify the patients according to more or less "typical" symptom combinations—which is to say, to formulate abstractions—we construct Table 8-10, which gives the number of different corresponding elements $\Delta_{i,j}$ that occur in each pair of columns. For instance, columns 1 and 2 differ only in row 5, so that $\Delta_{1,2} = 1$, while columns 1 and 9 differ in rows 1, 2, 4, 5, 8, and 9, whence $\Delta_{1,9} = 6$, etc. The table is of course symmetric: that is, $\Delta_{i,j} = \Delta_{i,j}$. The $\Delta_{i,j}$ can be considered as the *distances* between pairs of columns. Note that in Table 8-10 we have drawn lines demarcating the three groups of

TABLE 8-10. NUMBER OF DIFFERING SYMPTOMS FOR EACH PAIR OF COLUMNS OF TABLE 8-9

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	1	2	2	5	6	6	6	4	5	5
2	1	0	2	3	1	4	5	5	5	5	6	6
3	1	2	0	3	3	6	7	5	7	5	6	6
4	2	3	3	0	4	5	6	6	6	6	7	5
5	2	1	3	4	0	5	6	6	6	4	5	5
6	5	4	6	5	5	0	1	1	1	5	6	6
7	6	5	7	6	6	1	0	2	2	6	5	7
8	6	5	5	6	6	1	2	0	2	6	7	7
9	6	5	7	6	6	1	2	2	0	4	5	5
10	4	5	5	6	4	5	6	6	4	0	1	1
11	5	6	6	7	5	6	5	7	5	1	0	2
12	5	6	6	5	5	6	7	7	5	1	2	0

closely *clustered* columns, namely, columns 1 through 5, 6 through 9, and 10 through 12. Within the first group, the most "typical" column is column 1, since within the group 1 to 5 it is *closest* to all the others.

Similarly, column 6 is most typical within the group 6 to 9, and column 10 is most typical within the group 10 to 12. Thus our abstract formulation resulted in three *diseases*, for which columns 1, 6, and 10, respectively, present the most "typical" symptom combinations. Such a system of abstraction formulations has actually been utilized in taxonomy, to assist in formulating taxonomic classifications; it has been suggested for use as illustrated in the determination of new diseases. After the classifications have been formulated, further analysis might result in the elimination of both *redundant* and *irrelevant* symptoms or characteristics from consideration.†

In all this we have assumed that the method leads to a solution (i.e., that a solution *exists*), which is reasonable for the definition of disease entities arising from different etiologies or for the definition of taxonomic categories that arise from different evolutionary branches. That is, we have assumed that there exists a "natural" solution, due to a basic underlying population structure, in which combinations of characteristics should represent an abstract, or conceptual, entity. However, it is probably more generally true that abstractions, or concepts, are formed by people with "usefulness" as the primary criteria. In such a case past experience—namely, the human heritage called civilization—will have already defined a large majority of the most useful concepts, and the problem changes from one of *finding* new abstractions to one of *learning* already recognized abstractions. For our purposes, we desire to *teach* the computer to recognize abstractions.

Much of the research accomplished in this latter area is concerned with teaching the computer to recognize patterns, usually letter characters; this research has proceeded under various names, including "neural-net simulation," "self-organizing nets," "perceptron studies," etc. The methods pursued can basically be characterized as shown in Fig. 8-27. In its simplest form, there is an input pattern senser, which could be, say, an array of photocells upon which the image of a letter is projected, each photocell being turned *on* when the image is light in its vicinity. The pattern senser is connected to a matrix of elements, variously called "neurons," "association units," "connection units," etc., which in turn are connected to a series of output units. There is a *threshold* θ associated with each of the neurons, such that, if for a given pattern on the input senser more than θ of the inputs to the neurons (from the senser

† If $E[S(1), \ldots, S(n), D(1), \ldots, D(m)]$ is the Boolean function that embodies medical knowledge—see Sec. 8-5—then a symptom $S(j)$ is called *redundant* if there exists a k such that

$$E \rightarrow [S(j) = S(k)]$$

and $S(j)$ is called *irrelevant* if it does not appear in any prime implicant of E—see Chap. 10.

photocells) are *on*, then the neuron is turned *on*. A neuron that is *on* has a certain value $V > 0$ associated with it; then each output unit has a threshold ϕ such that, if the sum of the values of the *on* neurons connected to it is greater than ϕ, then the output unit will go *on*. The thresholds and values are all adjustable, and the problem is to choose them in such a way, say, that, when an **A** is projected on the senser, only output unit 1 will go *on;* when a **B** is projected, only output unit 2 will go *on;* and so forth. The system is said to have formed an abstraction of an **A**, or of a **B**, etc., if it can recognize all kinds of **A**'s, whether

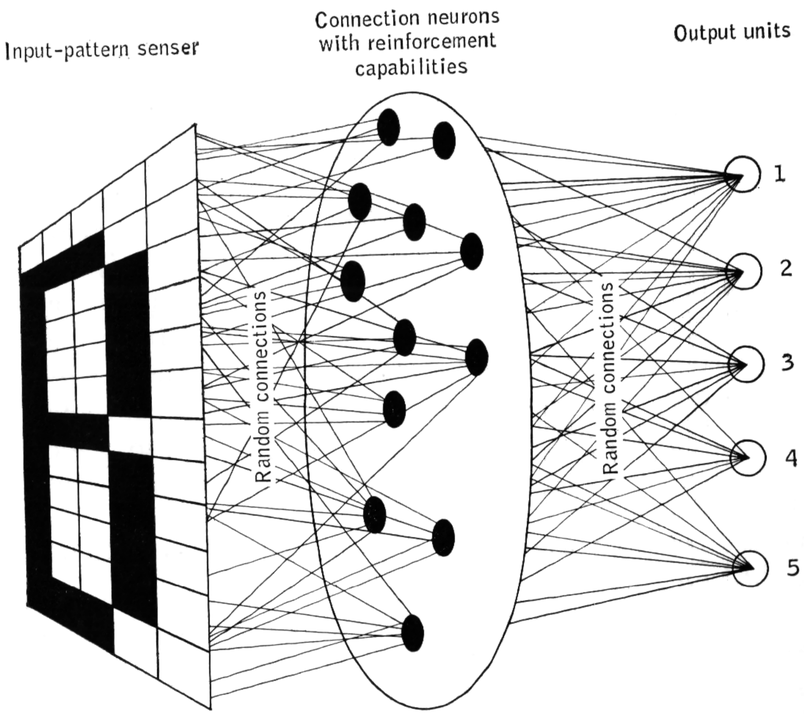


FIG. 8-27. Technique for teaching the computer to recognize abstractions.

they be large, small, crooked, rotated, translated, etc., as distinguished from all kinds of **B**'s, and so forth. The system is *taught* the abstractions by repeated applications of the following "reinforcement rules": An **A** is projected on the senser; if output unit 1 is not *on*, the thresholds of all *on* neurons connected to output unit 1 are lowered, and their values raised; if any other output unit is *on*, the thresholds of all *on* neurons connected to it are raised as their values are lowered. After many repetitions of this process for the various forms of **A**, the system is similarly "conditioned" for **B** with unit 2 as the associated output unit; and so forth. There are many variations to this scheme, using different reinforcement rules, having more than one level of connection neurons, including feedback connections between neurons, and so forth.

Almost all pattern-recognition methods that include learning use the basic ingredients of the two methods described: namely, selection through some form of characteristic matching, and value reinforcement guided by iterated sampling. The difference between the two methods is that the former is set up to determine the concepts or categories from a given set of characteristic specifications, whereas the latter is primarily devoted to determining a good set of characteristic specifications given the concepts or categories.

However, none of these methods relate to the syntactical type of concept formation. This is because the syntactical type of abstraction embodies a building-block procedure in which previous results are utilized in a hierarchical and recursive system. Apparently, deductive inference cannot contend with such a system, and inductive inference is required.

The use of inductive inference in the formulation of abstractions also involves two types of problems: first, that of *determining which syntactical structures should be defined given the relationships among the entities* of the system under consideration; and second, that of *determining the relationships between already defined syntactical structures.* The former problem is primarily concerned with the criteria of usefulness to be applied to the system; the latter is the problem that the four-year-old boy faces when his parents point out to him "houses," "windows," "roofs," "chimneys," "doors," etc. In both cases, trial and error are required, guided by heuristic procedures that may be built into the human brain, or acquired, or both. The frontiers of current research efforts are in these areas, and further consideration of these interesting points is beyond the scope of our present treatment.

Creativity. Creativity, whether it be in music or in art, in mathematics or in physics, does not occur in a vacuum, but rather is affected by historical precedent. It is perhaps as much a product of the age in which it arises as it is of the mind of the individual from whom it springs; although it embodies the most personal of efforts, it derives its sustenance from the society in which it develops. Creativity produces structure out of disorder, form out of chaos, but the structure and form must meet aesthetic requirements as well. Such attributes, among many others, must be encompassed by a computer program that is to achieve creativity. In our discussion of creativity, we shall first present an example of musical composition,† to illustrate the nature of at least one approach to the problem, and then give a brief critique of the method utilized.

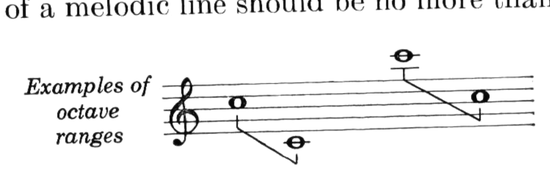
Music consists of tone sequences and the rhythm in which they occur—and both these aspects must be considered by a computer program that is

† The illustration is taken from L. A. Hiller, Jr., and L. M. Isaacson, "Experimental Music," McGraw-Hill Book Company, Inc., New York, 1959.

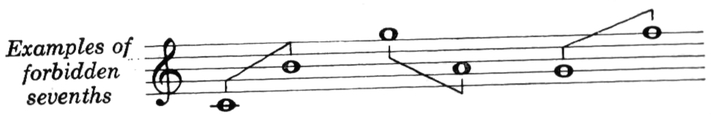
to "compose" music. In the illustration to be described, the underlying idea is to generate *random* notes and *random* tune-interval patterns, but to *select* from these only those that meet the specifications, or constraints, of certain rules of melody and harmony for the notes and of rhythm for the time-interval patterns. Thus the problem becomes one of formulating the *rules of selection* so that the resulting sequence of tones and rhythms will be a musical composition. We shall consider first some of the rules of melody and harmony used in the illustration, and then rules for the selection of the rhythms.

In the historical development of musical composition, the rules of *counterpoint* were set forth by Palestrina, a pre-Bach composer of choral religious music; the study of musical composition almost always begins with a course in counterpoint. These rules are chosen as the selection constraints for melody and harmony (the latter for four voices). There are three types of rules: those for melody alone—i.e., the horizontal sequence of tones; those for harmony alone—i.e., the vertical combination of tones that will occur simultaneously; and combined rules referencing both melody and harmony. It is not appropriate here to give more than just a few samples of each type of rule, although 16 rules in all were used.

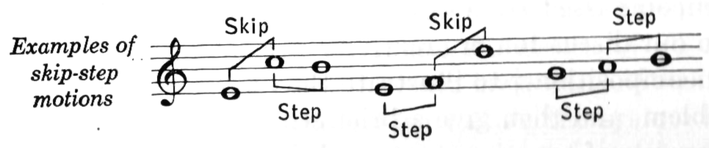
1. SAMPLES OF MELODIC RULES. The range from the lowest note to the highest note of a melodic line should be no more than an octave: e.g.,



A melodic skip of a major or minor seventh is forbidden: e.g.,

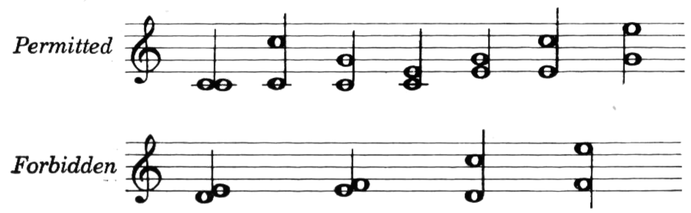


Any melodic skip, or movement of a minor third or more, must be followed by a tone repeat or by a step, i.e., a movement of a major or minor second; a melodic step may be followed by a step or by a skip: e.g.,

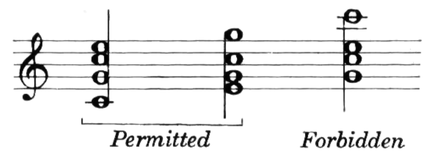


No more than one successive repetition of a given note is permitted.

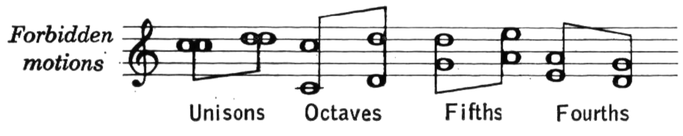
2. SAMPLES OF HARMONIC RULES. Permitted intervals include unisons, octaves, perfect fifths, and major and minor thirds and sixths. Forbidden intervals include major and minor seconds and sevenths, augmented fourths, and diminished fifths: e.g.,



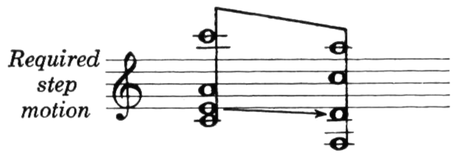
The perfect fourth is permitted only if it does not occur between an upper voice and the voice sounding the lowest note of a chord: e.g.,



3. SAMPLES OF COMBINED RULES. Parallel unisons, parallel perfect fifths and fourths, and parallel octaves are forbidden: e.g.,



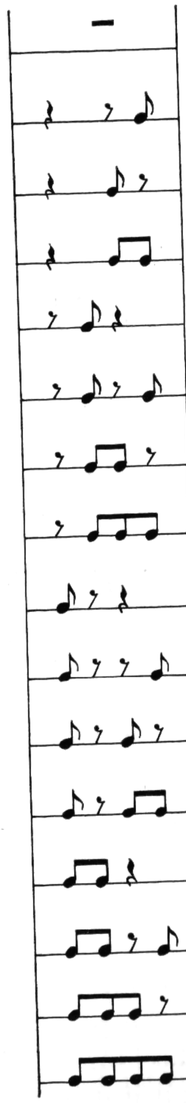
In proceeding from one chord to the next, at least one of the four voices must move by a step motion or remain stationary: e.g.,



The entire collection of rules required that the processing be accomplished in three stages, for the initial notes, the intermediate notes, and the final, or cadence, notes. The process progresses by repeatedly generating a random note and checking it with the rules, recording it if it is permitted and rejecting it if it is forbidden. When there are only a limited number of possible next notes, a random number determines which is chosen. The number of notes as well as voices in a melodic line is predetermined, and from this the computer program recognizes the end of each such line, for which it is to utilize the cadence rules.

To accomplish the introduction of rhythm into the melody, a $\frac{4}{8}$ meter—four eighth notes to the measure—was used. If no shorter than an eighth note is utilized, then a four-bit binary number will represent all the 16 rhythmic patterns available for a single measure (see Table 8-11). Thus

TABLE 8-11. BASIC RHYTHMIC SCHEME FOR $\frac{4}{8}$ METER

Binary number	*Rhythm*
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

random numbers could be utilized to generate rhythms. However, a common characteristic of music is that there exist correlations of rhythms between voices in a polyphonic piece of music. For four voices, another four-bit number was used: 1111 meant that all four voices would have the same rhythm for that measure, 0101 that voices 2 and 4 would have the same rhythm for the measure, etc. The numbers 0001, 0010, 0100 were used to signify that the pairs of voices 1-2 and 3-4, 1-3 and 2-4, and 1-4 and 2-3, respectively, would have the same rhythms. In addition,

another number, from 0 to 15, was used to determine for how many measures a particular rhythmic pattern would be repeated. *All three* of these numbers were chosen randomly.

There are of course many further details to the work, including the use of probability distributions for selecting the various random notes and numbers, and so forth. In any event, we have tried to give a general idea of the approach taken in one instance of writing a computer program for the creation of a musical composition. The question arises as to whether this is really creativity. Although the experiment is exceedingly interesting, in the author's opinion there is a lack of some of the necessary ingredients of creativity:† over-all planning or direction is missing, leading to a sense of incompleteness; the resulting music is digressive; it lacks symmetry and the recursive building up of ideas.

Two problems may be distinguished—that of imparting form, direction, and unity to a particular musical composition, and that of comprehending an over-all, across-the-board characterization of a "style," i.e., an abstraction that is recognizable in collections of compositions of a single composer. It is well known, for example, that the change in character of a composer's works can often be directly related to the influences of other composers, teachers, etc., with whom he associated or studied. Can a computer be programmed to grasp such abstractions in a usable form? Can a computer program be written that will "mix" abstractions, with the generation of new concepts? These are some of the questions that concern modern programming research. The answers will probably involve the notion of syntactical concept formation, giving the computer the ability to comprehend musical abstractions and to use such abstractions as a guide to creativity.

8-9. Exercises

8-1. Draw a detailed flow chart of a rummy card game.

8-2. Define a collection of two-address list-processing instructions sufficient for programming the flow chart of Exercise 8-1.

8-3. Write a code using the instructions of Exercise 8-2 for the manipulation aspects of the flow chart of Exercise 8-1. Omit detailed consideration of any strategy considerations.

8-4. If a one-address symbolic system of list-processing instructions were to be defined, how would the multiaddress aspects of a list-processing instruction be handled? (HINT: Use push-down lists.)

Apply the translator to the following ALGOL statements using the syntax given in Table 8-4:

8-5. A:=((B+C)×D)+E

8-6. A:=(B↑C↑(D+E)×B+C/(D+C))×E

† Based on hearing excerpts from the "Illiac Suite," composed by the Illiac computer at the University of Illinois using Hiller and Isaacson's program.

8-7. How can numbers be incorporated into the syntax of Table 8-4?

8-8. Using the extended syntax table of Exercise 8-7, apply the translator to

A:=−B+C↑2−4×A×C

Apply the translator to the following ALGOL statements using the syntax given in Table 8-5:

8-9. A:=((B+C)×D)+E

8-10. A:=(B↑C↑(D+E)×B+C/(D+C))×E

8-11. A:=(D+C/(B×A↑B↑(C+D)))×E

8-12. Compare the target-language codes produced in Exercises 8-5 and 8-6 with those produced in Exercises 8-9 and 8-10, and explain why the differences occur.

8-13. Write a syntax table for the translation of a simple Boolean expression of ALGOL as the source language to a one-address symbolic machine language as the target language. (Define instructions in the target language as needed.)

8-14. Using the syntax table generated in Exercise 8-13, apply the translator to

A:=B∧(C∨D)⊃E≡F

8-15. Suppose food F_1 contains 2 units/lb of vitamin A, 5 units/lb of vitamin B, and costs \$1 per pound; food F_2 contains 4 units/lb of vitamin A, 2 units/lb of vitamin B, and costs 80 cents per pound; and food F_3 contains 3 units/lb of vitamin A, 5 units/lb of vitamin B, and costs \$1.50 per pound. What diet of these three foods will contain at least 80 units of vitamin A and 60 units of vitamin B and be of minimum cost?

8-16. Carry out the actual computations that resulted in Fig. 8-14.

8-17. Solve the following game-theoretical problem (reproduced from J. D. Williams, "The Compleat Strategyst," McGraw-Hill Book Company, Inc., New York, 1954):

Frank is hurrying home late, after a particularly grueling day, when it pops into his mind that today is Kitty's birthday! Or is it? Everything is closed except the florist's. If it is not her birthday and he brings no gift, the situation will be neutral, i.e., value 0. If it is not and he comes in bursting with roses, and obviously confused, he may be subjected to the Martini test, but he will emerge in a position of strong one-upness—which has value 1. If it is her birthday and he has, clearly, remembered it, that is worth somewhat more, say, 1.5. If he has forgotten it, he is down like a stone, say, −10. What should Frank do?

Using the conditional probabilities of Table 8-12 in conjunction with Fig. 8-15, make the differential diagnosis for patients presenting the following symptoms:

8-18. $S(1) \cdot S(2)$.

8-19. $S(1) \cdot S(2) \cdot \overline{S(3)}$.

8-20. $S(1) \cdot \overline{S(2)} \cdot \overline{S(3)}$.

8-21. $\overline{S(1)} \cdot \overline{S(2)} \cdot S(3)$.

8-22. Perform the calculations that lead to Fig. 8-19.

Apply the general problem-solving program to proving the following trigonometric identities:

8-23. $\sec^2 \theta - \tan^2 \theta = 1$.

8-24. $\csc^2 \theta = 1 + \cot^2 \theta$.

8-25. $\cos \theta - \cos \phi = -2 \sin \frac{1}{2}(\theta + \phi) \sin \frac{1}{2}(\theta - \phi)$. (HINT: If the operators given in the text are not sufficient to enable this proof, what additional operators would you add?)

TABLE 8-12. CONDITIONAL AND TOTAL PROBABILITIES

Disease complex subscript j \ Symptom complex superscript i	0	1	2	3	4	5	6	7	$P(d_j)$
0	0	0	0	0	0	0	0	0	0.000
1	0	0.167	0	0	0.333	0.500	0	0	0.600
2	0	0.067	0.067	0.200	0.067	0.167	0.167	0.265	0.150
3	0	0	0	0	0	0.333	0	0.667	0.150
4	0	0.100	0	0	0.300	0.600	0	0	0.050
5	0	0	0	0	0	1.000	0	0	0.040
6	0	0	0	0	0.200	0.200	0.200	0.400	0.005
7	0	0	0	0	0	0.200	0	0.800	0.005

$P(s^i|d_j)$

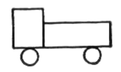
8-26. Using *deductive* inference, show that the picture on the right of Fig. 8-26*a* is a house.

8-27. Using *inductive* inference, show that the picture on the right of Fig. 8-26*a* is a house.

8-28. Compose a set of syntactical definitions for a toy railroad train.

8-29. Combine the definitions of Exercise 8-28 with those of the definition of a house given in the text, and try examples of distinguishing a house from a railroad train by deductive inference.

8-30. Construct a set of operators for the concept of a toy railroad train, where the concept is



8-31. Combine the operators of Exercise 8-30 with those given in the text for the concept of a house, and try examples of distinguishing a house from a railroad train by inductive inference.

8-32. Use the rules of musical composition as given in the text, and compose a musical composition by throwing dice.

REFERENCES

List Processing

Barnett, M. P.: A FORTRAN Encoded Symbol Pattern Locator, privately circulated "ditto," 1959.

Carr, J. W., III: Recursive Subscripting Compilers and List-type Memories, *Communs. ACM*, vol. 2, pp. 4–6, February, 1959.

Gelernter, H., J. R. Hansen, and C. L. Gerberich: A Fortran-compiled List-processing Language, *J. Assoc. Computing Mach.*, vol. 7, pp. 87–101, April, 1960.

Green, B. F.: IPL-V: The Newell-Shaw-Simon Programming Language, *Behavioral Sci.*, vol. 5, no. 1, pp. 94–98, 1960.
———: Computer Languages for Symbol Manipulation, *IRE Trans. on Human Factors in Electronics*, vol. HFE-2, no. 1, pp. 3–7, March, 1961.
Green, Julien: Symbol Manipulation in XTRAN, *Communs. ACM*, vol. 3, no. 4, pp. 213–214, April, 1960.
McCarthy, John: Recursive Functions of Symbolic Expressions and Their Computation by Machine, *Communs. ACM*, vol. 3, no. 4, pp. 184–195, April, 1960.
Newell, A., and F. Tonge: An Introduction to Information Processing Language V, *Communs. ACM*, vol. 3, no. 4, pp. 205–211, April, 1960.
———, ———, E. A. Feigenbaum, G. H. Mealy, N. Saber, B. F. Green, and H. K. Wolfe: Information Processing Language V Manual, sec. I: The Elements of IPL Programming, *Paper* P-1897; sec. II: Programmer's Reference Manual, *Paper* P-1918, Rand Corp., Santa Monica, Calif., 1960.
Oswald, V. A., and R. H. Lawson: An Idioglossary for Mechanical Translation, *Mod. Language Forum*, vol. 38, no. 2, pp. 1–11, September–December, 1953.
Perlis, A. J., and C. Thornton: Symbol Manipulation by Threaded Lists, *Communs. ACM*, vol. 3, no. 4, pp. 195–205, April, 1960.

Automatic Programming Translation

Arden, B. W., B. A. Galler, and R. M. Graham: The Internal Organization of the MAD Translator, *Communs. ACM*, vol. 4, no. 1, pp. 28–30, January, 1961.
England, D., and E. Clark: The CLIP Translator, *Communs. ACM*, vol. 4, no. 1, pp. 19–22, January, 1961.
Evans, A., A. J. Perlis, and H. VanZoeren: The Use of Threaded Lists in Constructing a Combined ALGOL and Machine-like Assembly Processor, *Communs. ACM*, vol. 4, no. 1, pp. 36–41, January, 1961.
Floyd, R. W.: An Algorithm for Coding Efficient Arithmetic Operations, *Communs. ACM*, vol. 4, no. 1, pp. 42–51, January, 1961.
Gorn, Saul: Specification Languages for Mechanical Languages and Their Processors—A Baker's Dozen, *Communs. ACM*, vol. 4, no. 12, pp. 532–542, December, 1961.
Ingerman, P. Z.: Thunks, pp. 55–58; Dynamic Declarations, pp. 59–60, *Communs. ACM*, vol. 4, no. 1, January, 1961.
Irons, E. T.: A Syntax Directed Compiler for ALGOL 60, *Communs. ACM*, vol. 4, no. 1, pp. 51–54, January, 1961.
Ledley, Robert S., and James B. Wilson: Automatic-programming-language Translation through Syntactical Analysis, *Communs. ACM*, vol. 5, no. 3, pp. 145–155, March, 1962.
Perlis, A., J. W. Smith, and A. Evans, Jr.: "TASS" (Technical Assembly System), Computation Center, Carnegie Institute of Technology, Pittsburgh, February, 1959.
Ross, P. T.: A Generalized Technique for Symbol Manipulation and Numerical Calculation, *Communs. ACM*, vol. 4, no. 3, pp. 147–150, March, 1961.
Steinbuch, K.: Automatische Zeichenerkennung, *Nachrtech. Z.*, vol. 11, pp. 210–219, 237–244, 1958.
Yngve, V.: A Programming Language for Mechanical Translation, *Mech. Transl.*, vol. 5, pp. 25–41, July, 1958.

Modern Mathematical Decision Theory

Bellman, R.: "Dynamic Programming," Princeton University Press, Princeton, N.J., 1957.

———: "Adaptive Control Processes," Princeton University Press, Princeton, N.J., 1961.
Bottenbuch, H., et al.: Applications of Logic to Advanced Digital Computer Programming, *Summer Session Notes*, University of Michigan, Ann Arbor, 1957.
Church, A.: "Introduction to Mathematical Logic," vol. 1, Princeton University Press, Princeton, N.J., 1956.
Gass, S. I.: "Linear Programming," McGraw-Hill Book Company, Inc., New York, 1958.
Howard, R. A.: "Dynamic Programming and Markov Processes," John Wiley & Sons, Inc., New York, 1960.
Kemeny, J. G., H. Mirkel, J. L. Snell, and G. L. Thompson: "Finite Mathematical Structures," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1959.
——— et al.: Symbolic Work on High Speed Computers, *Dartmouth Math. Project Rept.* 4, June, 1959.
Kleene, S. C.: "Introduction to Metamathematics," D. Van Nostrand Company, Inc., Princeton, N.J., 1952.
Magnus, K.: Nonlinear Problems of Control Theory, *Math. Tech. Wirtsch.*, no. 1, pp. 19–25, January, 1960.
Polya, G.: "Mathematics and Plausible Reasoning," Princeton University Press, Princeton, N.J., 1954.
Shepard, R. N., C. I. Hovland, and H. M. Jenkins: Learning and Memorization of Classifications, *Psychol. Monogr.*, 1961.
Williams, J. D.: "The Compleat Strategyst," McGraw-Hill Book Company, Inc., New York, 1954.

Computer Aids to Medical Diagnosis

Ledley, R. S., and L. B. Lusted: Electronic Computer Aids to Medical Diagnosis, *Proc. Second Intern. Conf. on Med. Electronics*, June, 1959, Paris, pp. 415–424.
——— and ———: Reasoning Foundations of Medical Diagnosis, *Science*, vol. 130, no. 3366, pp. 9–21, July 3, 1959.
——— and ———: The Use of Electronic Computers to Aid Medical Diagnosis, *Proc. IRE* (Medical Electronics Issue), vol. 47, no. 11, pp. 1970–1977, November, 1959.
——— and ———: The Use of Electronic Computers in Medical Data Processing: Aids in Diagnosis, Current Information Retrieval, and Medical Record Keeping, *IRE Trans. on Med. Electronics*, vol. ME-7, pp. 31–47, January, 1960.
——— and ———: Mathematical Models in Medical Diagnosis, *J. Med. Educ.*, vol. 35, no. 3, pp. 214–222, March, 1960.
——— and ———: Medical Diagnosis and Modern Decision Making, *Proc. Appl. Math. Symposium*, New York, Apr. 5–7, 1961.
——— and ———: Medical Diagnosis and Modern Decision Making, *Proc. Fourth Ann. Conf. on Med. Electronics Combined with the Fourteenth Ann. Conf. on Elec. Tech. in Med. Biol.*, New York, July, 1961.
———and ———: The Role of Computers in Medical Diagnosis, *Med. Dok.*, vol. 5, no. 3, pp. 70–78, July, 1961.
Warner, H. R., A. F. Toronto, L. G. Veasey, and R. Stephenson: A Mathematical Approach to Medical Diagnosis, *J. AMA*, vol. 177, no. 3, July 22, 1961.

Heuristic Programming

Bernstein, A., et al.: A Chess-playing Program for the IBM 704, *Proc. Western Joint Computer Conf.*, May, 1958, pp. 157–159.

Feigenbaum, E.: An Information Processing Theory of Verbal Learning, *Paper* P-1817, Rand Corp., Santa Monica, Calif., October, 1959.

———: The Simulation of Verbal Learning Behavior, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 121–132.

Friedberg, R. M.: A Learning Machine, pt. I, *IBM J. Research Develop.*, vol. 2, pp. 2–13, January, 1958.

Gelernter, H.: Realization of a Geometry Theorem-proving Machine, *Proc. Intern. Conf. on Inform. Processing*, pp. 273–282, UNESCO, Paris, 1960.

———, J. R. Hansen, and D. W. Coveland: Empirical Explorations of the Geometry Theorem Machine, *Proc. Western Joint Computer Conf.*, May, 1960, pp. 143–150.

——— and N. Rochester: Intelligent Behavior in Problem Solving Machines, *IBM J. Research Develop.*, vol. 2, no. 4, pp. 336–345, October, 1958.

Gilmore, P.: A Program for the Production of Proofs for Theorems Derivable within the First Order Predicate Calculus from Axioms, *Proc. Intern. Conf. on Inform. Processing*, 1959.

Green, B. F., A. K. Wolf, and C. Chomsky: Baseball: An Automatic Question-answerer, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 219–224.

Kister, J., et al.: Experiments in Chess, *J. Assoc. Computing Mach.*, vol. 4, no. 2, pp. 174–177, April, 1957.

Kochen, M.: Experimental Study of Hypothesis Formulation by Computer, *IBM Research Rept.* RC-294, IBM Research Center, Yorktown Heights, N.Y., 1960.

Lawrence, D. H., and G. R. Coles: Accuracy of Recognition with Alternatives before and after the Stimulus, *J. Exptl. Psychol.*, vol. 47, pp. 208–214, 1954.

McCarthy, J.: The Advice Taker, *Proc. Symposium on Mechanization of Thought Processes*, Nov. 24–27, 1958, National Physical Labouratory, Teddington, England.

———: Programs with Common Sense, *Proc. Symposium on Mechanization of Thought Processes*, Nov. 24–27, 1958, National Physical Labouratory, Teddington, England, vol. 1, pp. 75–84.

Minsky, M.: Descriptive Languages and Problem Solving, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 215–218.

Newell, A., and J. C. Shaw: Empirical Exploration of the Logic Theory Machine, *Proc. Western Joint Computer Conf.*, February, 1957, pp. 218–230.

——— and ———: Programming the Logic Theory Machine, *Proc. Western Joint Computer Conf.*, February, 1957, pp. 230–240.

———, ———, and H. A. Simon: Chess Playing Programs and the Problem of Complexity, *IBM J. Research Develop.*, vol. 2, pp. 320–335, October, 1958.

———, ———, and ———: Report on a General Problem-solving Program, *Paper* P-1584, Rand Corp., Santa Monica, Calif., February, 1959.

———, ———, and ———: A Variety of Intelligent Learning in a General Problem Solver, *Paper* P-1742, Rand Corp., Santa Monica, Calif., July, 1959.

——— and H. A. Simon: The Logic Theory Machine, *IRE Trans. on Inform. Theory*, vol. IT-2, pp. 61–79, September, 1956.

Oettinger, A. G.: Simple Learning by a Digital Computer, *Proc. Assoc. Computing Mach.*, September, 1952.

Samuel, A. L.: A Checker Playing Program, *IBM J. Research Develop.*, vol. 3, no. 3, pp. 210–229, July, 1959.

Shaw, J. C., A. Newell, H. A. Simon, and T. O. Ellis: A Command Structure for Complex Information Processing, *Proc. Western Joint Computer Conf.*, May, 1958, pp. 119–128.

Simon, H. A.: Modeling Human Mental Processes, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 51–62.

Tonge, F. M.: Summary of a Heuristic Line Balancing Procedure, *Paper* P-1799, Rand Corp., Santa Monica, Calif., 1959.

Wang, Hao: Toward Mechanical Mathematics, *IBM J. Research Develop.*, vol. 4, no. 1, pp. 2–22, January, 1960.

Pattern Recognition and Concept Formation

Alluisi, E. A.: Conditions Affecting the Amount of Information in Absolute Judgments, *Psychol. Rev.*, vol. 64, pp. 97–103, 1957.

Amarel, S.: An Approach to Automatic Theory Formation, paper presented at the Illinois Symposium on the Principles of Self Organization, 1960.

Baran, P., and G. Estrin: An Adaptive Character Reader, *IRE WESCON Conv. Record*, vol. 4, pt. 4, pp. 29–41, 1960.

Bartlett, F. C.: "Thinking," Basic Books, New York, 1958.

Baxendale, P. B.: Machine-made Index for Technical Literature—An Experiment, *IBM J. Research Develop.*, vol. 2, no. 4, pp. 354–361, October, 1958.

Clark, W. A. and B. G. Farley: Generalization of Pattern Recognition in a Self-organizing System, *Proc. Western Joint Computer Conf.*, 1955, pp. 86–91.

Culbertson, J. T.: "Consciousness and Behavior," Wm. C. Brown, Dubuque, Iowa, 1950.

Dinneen, G. P.: Programming Pattern Recognition, *Proc. Western Joint Computer Conf.*, March, 1955, pp. 94–100.

Doyle, W.: Recognition of Sloppy, Hand-printed Characters, *Proc. Western Joint Computer Conf.*, 1960, pp. 133–142.

Edmundson, H. P., and R. E. Wyllys: Automatic Abstracting and Indexing—Survey and Recommendations, *Communs. ACM*, vol. 4, no. 5, pp. 226–234, May, 1961.

Grimsdale, R. L., F. H. Sumner, C. J. Tunis, and T. Kilburn: A System for the Automatic Recognition of Patterns, *Proc. IEE* (*London*), pt. B, vol. 106, pp. 210–221, March, 1959.

Hebb, D. O.: "The Organization of Behavior," John Wiley & Sons, Inc., New York, 1959.

Highleyman, W. H., and L. A. Kamentsky: Comments on a Character Recognition Method of Bledsoe and Browning, *IRE Trans. on Electronic Computers*, vol. EC-9, p. 263, June, 1960.

Hiller, L. A., Jr., and L. M. Isaacson: "Experimental Music," McGraw-Hill Book Company, Inc., New York, 1959.

Hovland, C. I., and E. B. Hunt: Computer Simulation of Concept Attainment, *Behavioral Sci.*, vol. 5, pp. 265–267, July, 1960.

——— and ———: Programming a Model of Human Concept Formulation, *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., pp. 145–156.

Hunt, E. B., and C. I. Hovland: Order of Consideration of Different Types of Concepts, *J. Exptl. Psychol.*, vol. 59, pp. 220–225, 1960.

Keller, H.: Finite Automata, Pattern Recognition, and Perceptrons, *AEC Computing Center and Appl. Math. Center Rept.* NYO-2884, 1960.

Luhn, H. P.: The Automatic Creation of Literature Abstracts, *IBM J. Research Develop.*, vol. 2, no. 2, pp. 159–165, April, 1958.

Miller, G. A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, *Psychol. Rev.*, vol. 63, pp. 81–96, 1956.

Minsky, M.: Steps toward Artificial Intelligence, *Proc. IRE* (Special Issue on Computers), vol. 49, no. 1, pp. 8–30, January, 1961.

Newell, A., J. C. Shaw, and H. A. Simon: The Processes of Creative Thinking, *Paper* P-1320, Rand Corp., Santa Monica, Calif., August, 1958.

——— and H. A. Simon: The Simulation of Human Thought, *Paper* P-1734, Rand Corp., Santa Monica, Calif., 1959.

Oswald, V. A., et al.: Automatic Indexing and Abstracting of the Contents of Documents, RADC-TR-59-208, pp. 5–34, 59–133, Oct. 31, 1959, prepared for the Rome Air Development Center, Air Research and Development Command, U.S. Air Force.

Pitts, W., and W. S. McCulloch: How We Know Universals: The Perception of Auditory and Visual Forms, *Bull. Math. Biophys.*, vol. 9, pp. 127–147, 1947.

Rath, G. J., A. Resnick, and T. R. Savage: The Formation of Abstracts by the Selection of Sentence, *Research Rept.* RC-184, IBM Research Center, Yorktown Heights, N.Y., June 29, 1959.

Resnick, A., and T. R. Savage: A Re-evaluation of Machine Generated Abstracts, *Research Rept.* RC-230, IBM Research Center, Yorktown Heights, N.Y., Mar. 1, 1960.

Roberts, L. G.: Pattern Recognition with Adaptive Network, *IRE Conv. Record*, vol. 8, pt. 2, pp. 66–70, 1960.

Rosenblatt, F.: The Perceptron, A Probabilistic Model for Information Storage and Organization in the Brain, *Psychol. Rev.*, vol. 65, pp. 386–408, 1958.

———: The Perceptron: A Theory of Statistical Separability in Cognitive Systems, *Cornell Aeronaut. Lab. Rept.* V6-1196-6-1, January, 1958.

———: Two Theorems of Statistical Separability in the Perceptron, *Proc. Symposium on Mechanization of Thought Processes*, Nov. 24–27, 1958, National Physical Labouratory, Teddington, England. Also *Cornell Aeronaut. Lab. Rept.* V6-1196-6-2, September, 1958.

———: "Perceptual Generalization over Transformation Groups in Self-organizing Systems," Pergamon Press, New York, 1959.

———: Perceptron Simulation Experiments, *Proc. IRE*, vol. 48, pp. 301–309, March, 1960.

Selfridge, O. G.: Pattern Recognition and Modern Computers, *Proc. Western Joint Computer Conf.*, March, 1955, pp. 91–93.

———: Pandemonium: A Paradigm for Learning, *Proc. Symposum on Mechanization of Thought Processes*, Nov. 24–27, 1958, National Physical Labouratory, Teddington, England, pp. 511–535.

——— and U. Neisser: Pattern Recognition by Machine and Men, *Sci. Am.*, vol. 203, no. 2, pp. 60–68, August, 1960.

Shepard, R. N., and J. J. Chang: Stimulus Generalization in the Learning of Classifications, *Bell Telephone Labs. Mim. Rept.*, 1961.

Sherman, H.: A Quasi-topological Method for the Recognition of Line Patterns, *Inform. Processing*, pp. 232–238, UNESCO, Paris, 1960.

Sperry, R. W.: Mechanisms of Neural Maturation, in S. S. Stevens (ed.), "Handbook of Experimental Psychology," pp. 236–280, John Wiley & Sons, Inc., New York, 1951.

Uhr, L.: Pattern Recognition—Computers as Models for Form Perception (draft), "ditto," 1960.

———: A Possibly Misleading Conclusion as to the Inferiority of One Method for Pattern Recognition to a Second Method to Which It Is Guaranteed to Be Superior, *IRE Trans. on Electronic Computers*, vol. EC-10, pp. 96–97, March, 1961.

——— and C. Vossler: Suggestions for Self-adapting Computer Models of Brain Functions, *Behavioral Sci.*, vol. 6, no. 1, pp. 91–97, January, 1961.

Unger, S. H.: Pattern Recognition and Detection, *Proc. IRE*, vol. 47, pp. 1737–1752, October, 1959.

Yovits, M. C., and S. Cameron (eds.): "Self-organizing Systems," Pergamon Press, New York, 1960.

PART 3

DATA-PROCESSING TECHNIQUES

CHAPTER 9

FUNDAMENTALS OF NUMERICAL ANALYSIS

Numerical analysis is a large and growing subject in itself, and our discussion must be limited to those elementary aspects which are considered of importance to the computer programmer. These questions naturally arise in the study of digital computers: If a computer can do little more than the elementary arithmetic operations, how can it be used to perform function evaluation, integration, or differentiation, or to solve ordinary and partial differential equations? Also, in performing long sequences of arithmetic operations, how is the accuracy of the computed results analyzed and controlled? We can give in this chapter no more than a brief introduction to the answers to these questions, but it is felt that such an elementary survey can well serve to orient the student in this field.

The chapter starts with a technique for solving linear simultaneous equations (Sec. 9-1), which arise frequently in all fields of science and engineering. Next, solution of algebraic and transcendental equations by means of successive approximations is considered (Sec. 9-2). Then the following three sections introduce the fundamental concept of polynomial approximations for the evaluation of functions. In passing, continued fractions are considered, so that the reader will not think there are no other methods for evaluating functions. The best-fit polynomial approximations are emphasized because of their extreme importance in digital-computer work. In order that the reader may develop a "numerical feeling" for such approximations, graphs are given of actually computed examples. The next five sections show how the same fundamental idea of polynomial approximation is used in numerical integration and differentiation and in the solution of differential equations. To complete the picture, the method of undetermined coefficients is given, so that the student can experiment and derive formulas for himself.

Up to this point discussion of error was omitted to allow full concentration on the concepts involved: the final section of this chapter is

concerned with accuracy and error. Our discussion will be limited to fundamental definitions and to consideration of the arithmetic operations. In performing a computation on a digital computer, the significance and accuracy associated with arithmetic operations are usually uppermost in the mind of the computer user. Hence an appreciation of the problems involved becomes of prime importance to the computer programmer.

9-1. Simultaneous Linear Equations

The Problem. Consider a set of n simultaneous linear equations in n unknowns $x_1, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \qquad (1)\\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \qquad (2)\\ &\cdots\cdots\cdots\cdots\cdots\cdots\\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= y_n \qquad (n) \end{aligned}$$

These equations admit of a unique nontrivial set of solutions,

$$x_i = x_i(a_{11}, \ldots, a_{nn}, y_1, \ldots, y_n)$$

if at least one y_i is not zero, and if the determinant $\det(a_{ij}) \neq 0$.† In order to describe a method for solving the equations, let us first consider a special set of equations with coefficients $s_{jk} = 1$ if $j = k$ and $s_{jk} = 0$ if $j > k$:

$$\begin{aligned} x_1 + s_{12}x_2 + s_{13}x_3 + s_{14}x_4 &= y_1\\ x_2 + s_{23}x_3 + s_{24}x_4 &= y_2\\ x_3 + s_{34}x_4 &= y_3\\ x_4 &= y_4 \end{aligned}$$

These four equations can easily be solved for the unknowns x_1, x_2, x_3, and x_4, for $x_4 = y_4$ is already given, and it follows that

$$\begin{aligned} x_3 &= y_3 - s_{34}x_4\\ x_2 &= y_2 - s_{24}x_4 - s_{23}x_3\\ x_1 &= y_1 - s_{14}x_4 - s_{13}x_3 - s_{12}x_2 \end{aligned}$$

Hence it becomes clear that, if we can somehow manipulate Eqs. (1), (2), . . . , (n) so that the diagonal coefficients become 1 and the coefficients below the diagonal become zero, then the solutions can be rapidly obtained.

The Method. This process can be accomplished as follows: First divide both sides of Eq. (1) by a_{11}. If $a_{11} = 0$, then rearrange the order

† See, for example, E. A. Guillemin, "The Mathematics of Circuit Analysis," pp. 13–18, John Wiley & Sons, Inc., New York, 1951.

of the equations so that $a_{11} \neq 0$. (This, of course, can always be done. Why?) This results in

$$1 \cdot x_1 + \frac{a_{12}}{a_{11}}x_2 + \cdots + \frac{a_{1n}}{a_{11}}x_n = \frac{y_1}{a_{11}} \qquad (1)^1$$

Multiply Eq. $(1)^1$ by a_{21}, and subtract the result from (2), obtaining

$$0 \cdot x_1 + b_{22}x_2 + \cdots + b_{2n}x_n = y_2^{(1)} \qquad (2)^1$$

where $b_{22} = a_{22} - \dfrac{a_{21}a_{12}}{a_{11}}$ $\cdots$ $b_{2n} = a_{2n} - \dfrac{a_{21}a_{1n}}{a_{11}}$

and $$y_2^{(1)} = y_2 - \frac{a_{21}y_1}{a_{11}}$$

Similarly, multiply $(1)^1$ by a_{31}, and subtract the result from (3), obtaining

$$0 \cdot x_1 + b_{32}x_2 + \cdots + b_{3n}x_n = y_3^{(1)} \qquad (3)^1$$

where $b_{32} = a_{32} - \dfrac{a_{31}a_{12}}{a_{11}}$ $\cdots$ $b_{3n} = a_{3n} - \dfrac{a_{31}a_{1n}}{a_{11}}$

and $$y_3^{(1)} = y_3 - \frac{a_{31}y_1}{a_{11}}$$

Continuing in this, we obtain a new set of Eqs. $(1)^1$, $(2)^1$, . . . , $(n)^1$ such that $b_{11} = 1$ and $b_{i1} = 0$ for $i > 1$. Of course, it is clear that our new set of equations has the same solution as the original set.

Next we start with $(2)^1$ and divide both sides by b_{22}. [If necessary, a rearrangement of the order of Eqs. $(2)^1$, . . . , $(n)^1$ can be made to ensure that $b_{22} \neq 0$.] We obtain

$$1 \cdot x_2 + \frac{b_{23}}{b_{22}}x_3 + \cdots + \frac{b_{2n}}{b_{22}}x_n = \frac{y_2^{(1)}}{b_{22}} \qquad (2)^2$$

Multiply $(2)^1$ by b_{32}, and subtract the result from $(3)^1$, obtaining

$$0 \cdot x_2 + c_{33}x_3 + \cdots + c_{3n}x_n = y_3^{(2)} \qquad (3)^2$$

where $c_{33} = b_{33} - \dfrac{b_{32}b_{23}}{b_{22}}$ $\cdots$ $c_{3n} = b_{3n} - \dfrac{b_{32}b_{2n}}{b_{22}}$

and $$y_3^{(2)} = y_3^{(1)} - \frac{b_{32}y_2^{(1)}}{b^{22}}$$

Similarly, we obtain

$$c_{43}x_3 + \cdots + c_{4n}x_n = y_4^{(2)} \qquad (4)^2$$

and the same for $(5)^2$, . . . , $(n)^2$. In this step we have made $c_{33} = 1$ and $c_{i2} = 0$ for $i > 2$.

Continuing in this way, we finally obtain, for the example of $n = 4$, the following equations, in the desired form for easy solution:

$$\begin{aligned} x_1 + \frac{a_{12}}{a_{11}} x_2 + \frac{a_{13}}{a_{11}} x_3 + \frac{a_{14}}{a_{11}} x_4 &= \frac{y_1}{a_{11}} \\ x_2 + \frac{b_{23}}{b_{22}} x_3 + \frac{b_{24}}{b_{22}} x_4 &= \frac{y_2^{(1)}}{b_{22}} \\ x_3 + \frac{c_{34}}{c_{33}} x_4 &= \frac{y_3^{(2)}}{c_{33}} \\ x_4 &= \frac{y_4^{(3)}}{d_{44}} \end{aligned}$$

Example. Let us carry this process through for a specific set of four equations.

$$\begin{aligned} 3x_1 + 9x_2 + 6x_3 - 12x_4 &= 9 && (1) \\ 2x_1 + 8x_2 + 12x_3 - 2x_4 &= 2 && (2) \\ 3x_1 + 12x_2 + 21x_3 - 9x_4 &= 15 && (3) \\ -x_1 - x_2 + 8x_3 + 8x_4 &= 5 && (4) \end{aligned}$$

Dividing both sides of (1) by 3, we find

$$x_1 + 3x_2 + 2x_3 - 4x_4 = 3 \qquad (1)^1$$

Now multiply Eq. $(1)^1$ by 2, and subtract from Eq. (2), obtaining

$$2x_2 + 8x_3 + 6x_4 = -4 \qquad (2)^1$$

Similarly, we multiply Eq. $(1)^1$ by 3 and subtract from Eq. (3), obtaining

$$3x_2 + 15x_3 + 3x_4 = 6 \qquad (3)^1$$

And finally $(4)^1$ is obtained by multiplying $(1)^1$ by -1 and subtracting from (4),

$$2x_2 + 10x_3 + 4x_4 = 8 \qquad (4)^1$$

Continuing, we form Eqs. $(2)^2$, $(3)^2$, and $(4)^2$,

$$\begin{aligned} x_2 + 4x_3 + 3x_4 &= -2 && (2)^2 \\ 3x_3 - 6x_4 &= 12 && (3)^2 \\ 2x_3 - 2x_4 &= 12 && (4)^2 \end{aligned}$$

Finally we have

$$\begin{aligned} x_3 - 2x_4 &= 4 && (3)^3 \\ 2x_4 &= 4 && (4)^3 \end{aligned}$$

and

$$x_4 = 2 \qquad (4)^4$$

Thus the set of equations from which the solution is obtained is

$$\begin{aligned} x_1 + 3x_2 + 2x_3 - 4x_4 &= 3 && (1)^1 \\ x_2 + 4x_3 + 3x_4 &= -2 && (2)^2 \\ x_3 - 2x_4 &= 4 && (3)^3 \\ x_4 &= 2 && (4)^4 \end{aligned}$$

whence

$$\begin{aligned} x_4 &= 2 && = 2 \\ x_3 &= 4 + 2 \times 2 && = 8 \\ x_2 &= -2 - 3 \times 2 - 4 \times 8 && = -40 \\ x_1 &= 3 + 4 \times 2 - 2 \times 8 - 3 \times (-40) && = 115 \end{aligned}$$

Figure 9-1 is the flow diagram for this method of solving simultaneous linear equations.

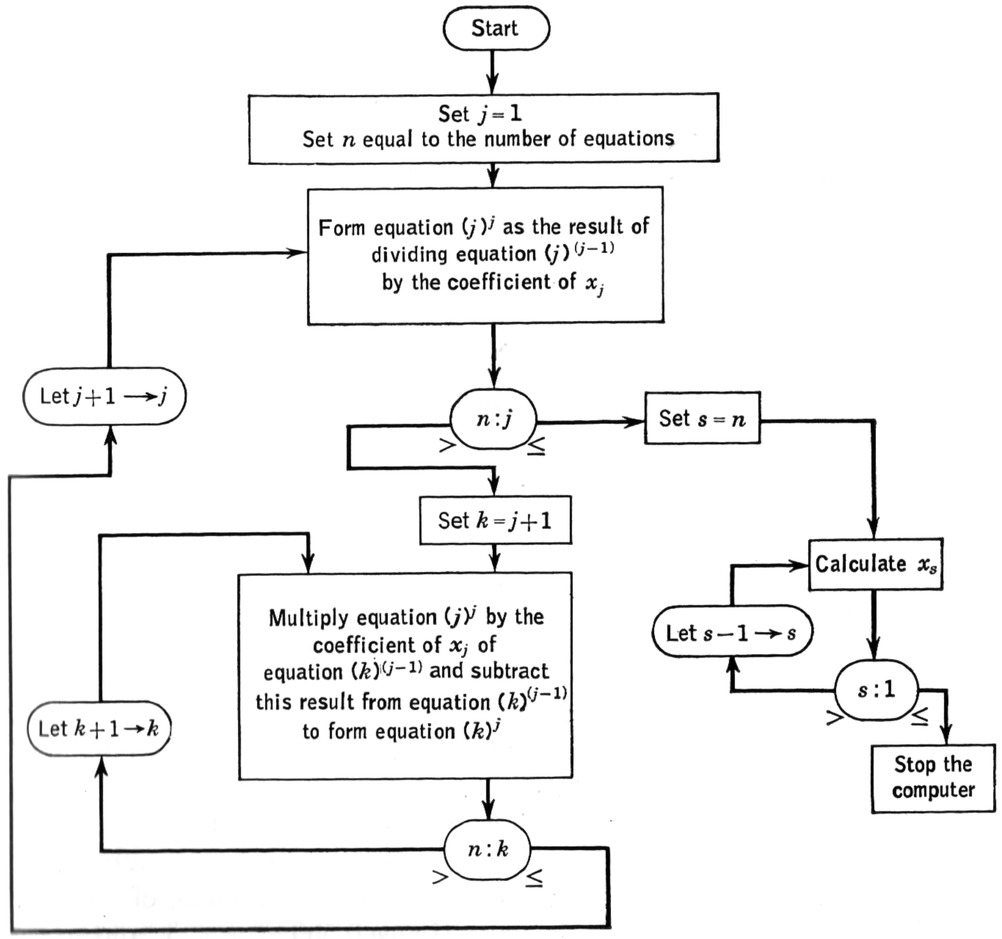


FIG. 9-1. Flow chart for simultaneous linear equations.

9-2. Algebraic and Transcendental Equations

Successive Approximations. The method of successive approximations can be used to evaluate the solutions of algebraic and transcendental

equations. Although the best method for making the successive approximations depends in any case on the nature of the functions involved, we shall try by means of specific examples to illustrate the general principles that apply. Often a graph of the functions involved in the equations will help in the determination of the best method of approximation.

Consider, for example, the equation

$$\sin x + 2x - 2 = 0$$

to be solved for x. We can write this as $\sin x = 2 - 2x$ and graph $y_T = \sin x$ and $y_P = 2 - 2x$ (see Fig. 9-2). The desired value of x is the

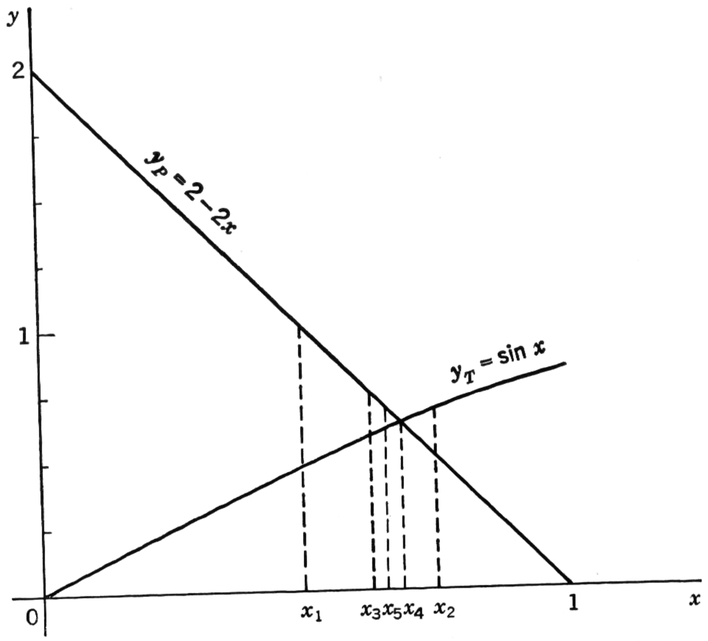


FIG. 9-2. Finding solution to $\sin x + 2x - 2 = 0$ by the method of successive approximations.

x coordinate of the intersection of the two curves, where $\sin x$ will be equal to $2 - 2x$. We know that the value of x must be between 0 and 1; hence we try x_1 such that $0 < x_1 < 1$. As can be seen from Fig. 9-2, if $\sin x_1 > 2 - 2x_1$, then the solution must lie to the left of x_1, that is, $x < x_1$; if $\sin x_1 < 2 - 2x_1$, then the solution must be to the right of x_1, that is, $x > x_1$. Suppose that we try $x_1 = 0.500$ (radians, of course); then $\sin x_1 = 0.479$, and $2 - 2x_1 = 1$. Hence $0.500 < x < 1$, and so we try $x_2 = 0.750$ (that is, half the distance between 0.500 and 1). Then $\sin x_2 = 0.682 > 2 - 2x_2 = 0.5$, whence $0.500 < x < 0.750$. We let $x_3 = 0.625$ (half the distance between 0.500 and 0.750); then

$$\sin x_3 = 0.585 < 2 - 2x_3 = 0.75$$

and so we try next 0.687, etc. In each step we find in which half of the previous interval the solution lies; Fig. 9-3 shows these calculations. The fact that we are halving each time means that we are developing the solution as a binary number. If the solution lies in the right half of an interval in a step, then a unit corresponds to this step in the solution; otherwise a zero corresponds. Hence the solution $x = .10101111$ according to Fig. 9-3, column y_P.

The flow diagram (see Fig. 9-4) is relatively simple: depending on whether $\sin x_i < 2 - 2x_i$ or $\sin x_i > 2 - 2x_i$, we determine the next trial solution x_{i+1} by adding to or subtracting from the previous trial solution x_i

(i)	x_i	$y_T = \sin x$	$y_P = 2 - 2x$	y_T greater	y_P greater
(1)	0.500	0.479	1		✓
(2)	0.750	0.682	0.5	✓	
(3)	0.625	0.585	0.75		✓
(4)	0.687	0.634	0.626	✓	
(5)	0.656	0.609	0.688		✓
(6)	0.672	0.621	0.656		✓
(7)	0.680	0.629	0.640		✓
(8)	0.684	0.631	0.632		✓

FIG. 9-3. Calculations for solving the equation $\sin x = 2 - 2x$.

one-half of the previously used increment Δx_i. Since each trial determines an additional bit of the solution, the procedure is repeated the same number of times as the number of bits of accuracy desired in the solution.

Accelerating the Convergence of Iterative Processes.† The problem of solving an equation $F(x) = 0$ can be stated in general as that of finding a value (root) of x, say x_r, such that $F(x_r) = 0$. The general *iteration algorithm* involves successive approximations $x_0, x_1, x_2, \ldots$, which are determined as follows:

1. If $F(x) = 0$ can be expressed in the form $x = f(x)$, then we let $x_{n+1} = f(x_n)$. For some initial guess x_0, we find $x_1 = f(x_0)$, $x_2 = f(x_1)$, $x_3 = f(x_2)$, . . . , continuing until, for some n, $|x_n - x_{n+1}|$ is less than some preassigned small number, the allowable error.

2. If $F(x) = 0$ cannot be written in the form of $x = f(x)$, then since as $n \to \infty$, $F(x_n) \to 0$ and $x_{n+1} - x_n \to 0$, we let $x_{n+1} = x_n + \theta F(x_n)$, where θ is some suitably chosen nonzero constant, and proceed as in case 1.

† This process was worked out by J. H. Wegstein and P. Henrici; see J. H. Wegstein, Accelerating Convergence of Iterative Processes, *Communs. ACM*, vol. 1, no. 6, pp. 9–13, June, 1958.

Let us consider case 1, for what we shall say also holds for case 2; Fig. 9-5 describes the process. Since $x = f(x)$, the object is to find the intersection of the two curves $y = x$ and $y = f(x)$. From $y = f(x_n)$ we

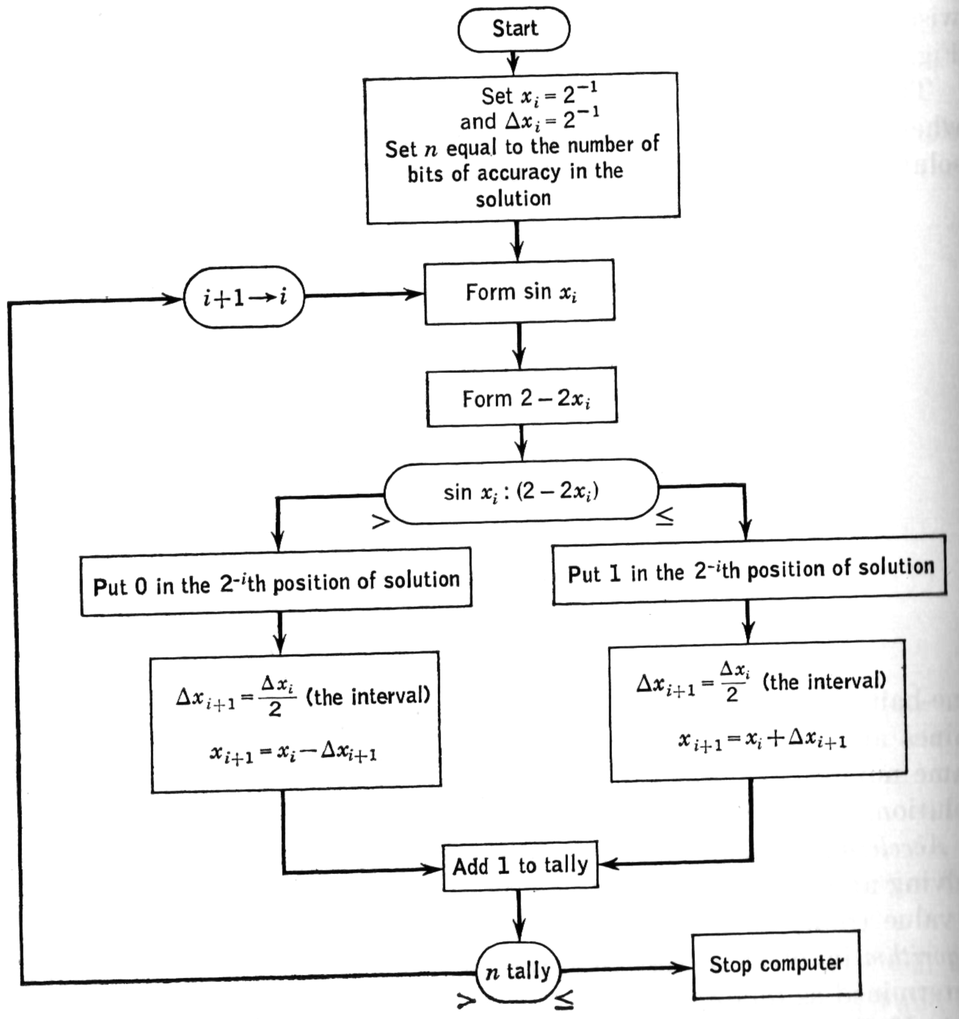


FIG. 9-4. Flow chart for solving the equation $\sin x = 2 - 2x$.

find y_n, point A; from $y = x$ we find $y_n = x_{n+1}$, point B; from $y = f(x_{n+1})$ we find y_{n+1}; from $y = x$ we find $y_{n+1} = x_{n+2}$; and so forth.

The iteration process just described can be accelerated, however, by observing from Fig. 9-5 that instead of taking x_{n+1} for the substitution into $f(x)$, we would do better to choose a value x^*_{n+1} for the substitution, where

$$x^*_{n+1} = qx_n + (1 - q)x_{n+1}$$

i.e., some value between x_n and x_{n+1}. For instance, in the case illustrated in Fig. 9-5, q should be chosen so that

$$\frac{q}{1 - q} = \frac{\overline{BC}}{\overline{AC}}$$

Of course we do not know the ratio $\overline{BC}/\overline{AC}$, but we can determine it approximately. For $\overline{PC} = \overline{BC}$, and thus

$$\frac{\overline{BC}}{\overline{AC}} = \frac{\overline{PC}}{\overline{AC}} = -m$$

where m is some value of $f'(x)$ between A and P (applying Rolle's theorem

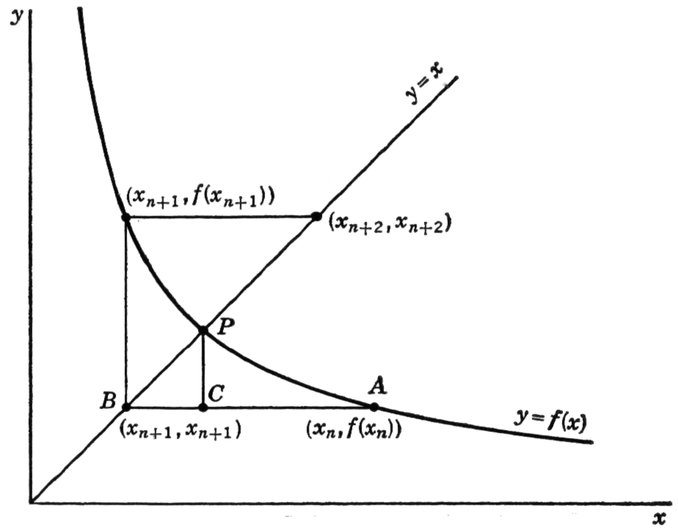


Fig. 9-5. The iterative process.

from the calculus). From the definition of the derivative, m can be approximated by

$$m \cong \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} = \frac{x_{n+1} - x_n}{x_n - x_{n-1}}$$

Hence we can take

$$\frac{q}{1 - q} = -m \qquad \text{or} \qquad q = \frac{m}{m - 1}$$

The iterative process can then be described by the flow chart of Fig. 9-6, wherein we must observe carefully when x^*_n or x_n is used.

As an example, consider the equation

$$x = \tfrac{1}{2}(e^{ax} - e^{-ax})$$

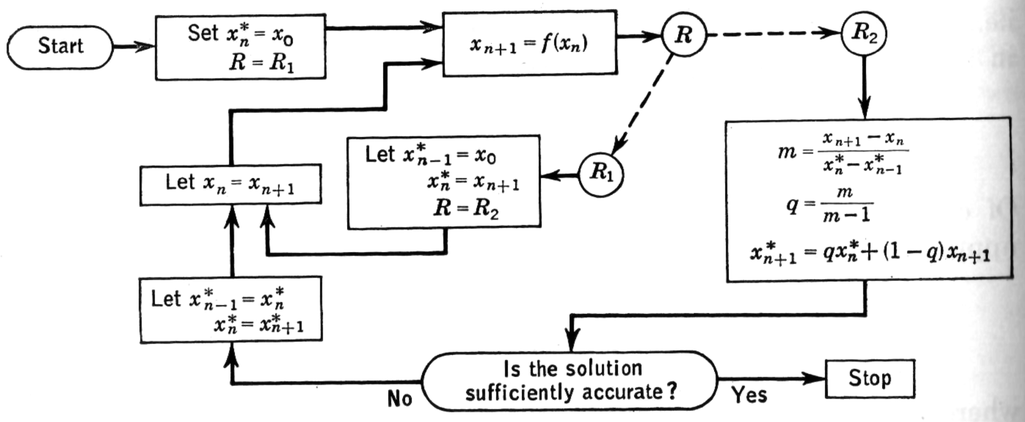


FIG. 9-6. Accelerating the convergence of the iterative process.

TABLE 9-1. EXAMPLE OF BOTH THE ORDINARY AND ACCELERATED ITERATIVE PROCESSES FOR THE ROOT $x = 0$ OF $x = \frac{1}{2}(e^{-0.5x} - e^{0.5x})$

n	x_n by ordinary iterative substitution	x_n by accelerating iterative substitution	Corresponding value of m
0	1.000	1.000	
1	−0.521	−0.521	
2	0.263	−0.00348	0.340
3	−0.132	-1.32×10^{-5}	0.335
4	0.066	-1.65×10^{-11}	0.333
5	−0.033		
6	0.017		
7	−0.008		
8	0.004		

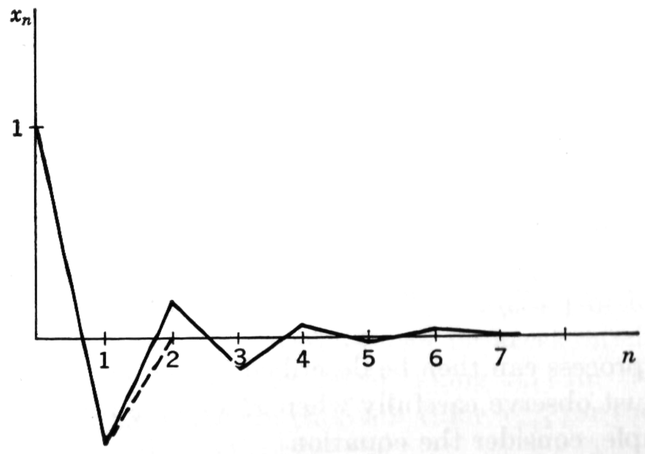


FIG. 9-7. Comparison of ordinary and accelerated (dashed line) iterative processes for the $x = 0$ root of $x = \frac{1}{2}(e^{-0.5x} - e^{0.5x})$.

With $\alpha = -0.5$, let us solve for the root $x = 0$, using both the ordinary and the accelerated iterative processes. Starting with $x_0 = 1$, we have Table 9-1 and Fig. 9-7.

The ordinary iterative process may result in four types of behavior. The successive values of x_n can (1) oscillate and converge, (2) oscillate and diverge, (3) converge monotonically, or (4) diverge monotonically. The accelerating method just presented will make the nonconverging types 2 and 4 converge and will accelerate the convergence of types 1 and 3. We have illustrated the situation for type 1. Exercises 9-9 to 9-11 will illustrate types 2 to 4.

9-3. Function Evaluation: Series and Continued Fractions

Series Approximations. In the previous section we have tacitly assumed that the values of the functions entering into the equations could somehow be obtained by the computer for any given value of x. For polynomials no problem arises, since the evaluation of a polynomial is easily coded. On the other hand, how are the values for sin x to be computed for a particular x? We shall discuss in this and the next two sections three ways in which such functions may be evaluated by the computer. These are (1) by power-series approximations, (2) by interpolation in a table, (3) by means of "best-fit" polynomial approximations. Of course our primary purpose is to discuss functions that are not themselves polynomials or fractions of polynomials, since the methods apply trivially for polynomials.

Many functions can be easily expanded in power series, and then the functions can be evaluated by means of these series for values of x within the region of convergence of the series. For example,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Then if sin 0.5 were required, the computer would be coded to compute

$$0.5 - \frac{0.5^3}{3!} + \frac{0.5^5}{5!} - \frac{0.5^7}{7!} + \cdots$$

If it were desired to know sin 0.5 to within ± 0.0001, then the series would be carried out until some term were less than 0.0001. For example, $\sin 0.5 = 0.5 - 0.02083333 + 0.00026042 - 0.00000155$, where we stop. Hence, to the desired accuracy, $\sin 0.5 = 0.4794$. Another way of interpreting this result is that the *polynomial approximation*

$$\sin^* x = x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

is sufficiently accurate for our purposes when $x = 0.5$. We write $\sin^* x$ instead of $\sin x$ to indicate that this is *just an approximation* to $\sin x$.

On the other hand, some series do not converge quite so rapidly—for example,

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

For $x = 0.5$ we find

$$\arctan^* 0.5 = 0.5 - 0.041666 + 0.006250 - 0.001116$$

which obviously is *not* accurate to ± 0.0001.

Often other expansions of a function can be found that might converge more rapidly than the power series; e.g., for arctan x we have the series

$$\arctan x = \frac{x}{1+x^2}\left[1 + \frac{2}{3}\frac{x^2}{1+x^2} + \frac{2\cdot 4}{3\cdot 5}\frac{x^4}{(1+x^2)^2} + \cdots\right]$$

Continued-fraction Approximations. There are other expansions of functions that often aid in their evaluation: for example, continued fractions such as

$$\arctan x = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{4x^2}{5 + \cfrac{9x^2}{7 + \cfrac{16x^2}{9 + \cdots}}}}}$$

It can be shown† that a continued fraction can be easily evaluated by means of 2×2 matrices as follows: Given the continued fraction

$$\frac{p}{q} = a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cfrac{b_5}{a_5 + \cdots}}}}$$

the *nth convergent* p_n/q_n is the result obtained when all terms from $n+1$ on are neglected; e.g., the fourth convergent is

$$\frac{p_4}{q_4} = a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4}}}$$

The nth convergent may be evaluated from the following matrix equation:

$$\begin{pmatrix} p_n & p_{n-1} \\ q_n & q_{n-1} \end{pmatrix} = \begin{pmatrix} a_1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} a_2 & 1 \\ b_2 & 0 \end{pmatrix}\begin{pmatrix} a_3 & 1 \\ b_3 & 0 \end{pmatrix} \cdots \begin{pmatrix} a_n & 1 \\ b_n & 0 \end{pmatrix}$$

† See L. M. Milne-Thomson, "The Calculus of Finite Differences," p. 108, The Macmillan Company, New York, 1951.

From this we can find p_n and q_n, and also p_{n-1} and q_{n-1}. For example, for arctan 0.5 we have

$$\begin{pmatrix} p_5 & p_4 \\ q_5 & q_4 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0.5 & 0 \end{pmatrix}\begin{pmatrix} 3 & 1 \\ 0.25 & 0 \end{pmatrix}\begin{pmatrix} 5 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 7 & 1 \\ 2.25 & 0 \end{pmatrix}$$
$$= \begin{pmatrix} 59.3750 & 8.00 \\ 128.0625 & 17.25 \end{pmatrix}$$

Hence

$$\frac{p_5}{q_5} = \frac{59.3750}{128.0625} = 0.4636 \qquad \text{and} \qquad \frac{p_4}{q_4} = \frac{8.00}{17.25} = 0.4637$$

Thus arctan 0.5 = 0.4636, to within ± 0.0001 (since the difference between the fourth and fifth convergents was 0.0001). We see that the infinite-fraction expansion of arctan x converged faster than its power-series expansion.

There are of course many other kinds of expansions of functions—for instance, Fourier-series expansions. However, the evaluation of a function by means of an expansion is not always the preferred method. Hence other means are often resorted to, and we shall consider a few of them briefly in the following sections.

Differences of a Polynomial. Since, as we shall see below, the polynomial approximation plays a central role in numerical analysis, it is important first to consider some very interesting properties of the polynomials themselves. These properties involve the so-called "differences of a polynomial." Consider a polynomial $y(x)$ whose values have been found at *equal* intervals $\Delta x_i = h$: that is, $y_0 = y(x_0)$, $y_1 = y(x_0 + h)$, $y_2 = y(x_0 + 2h)$, ... The differences of successive values of y, namely,

$$\Delta y_i = y_{i+1} - y_i = y(x_{i+1}) - y(x_i)$$

are called the *first differences* of the polynomial. The differences of successive first differences are called *second differences:*

$$\Delta^2 y_i = \Delta y_{i+1} - \Delta y_i = y_{i+2} - 2y_{i+1} + y_i$$

Similarly we can define *third differences, fourth differences,* and so on:

$$\Delta^3 y_i = \Delta^2 y_{i+1} - \Delta^2 y_i = y_{i+3} - 3y_{i+2} + 3y_{i+1} - y_i$$
$$\Delta^4 y_i = \Delta^3 y_{i+1} - \Delta^3 y_i = y_{i+4} - 4y_{i+3} + 6y_{i+2} - 4y_{i+1} + y_i$$

In general, the jth difference is given by

$$\Delta^j y_i = y_{i+j} - \binom{j}{1} y_{i+j-1} + \binom{j}{2} y_{i+j-2} - \cdots + (-1)^j y_i$$
$$= \sum_{k=0}^{j} (-1)^k \binom{j}{k} y_{i+j-k}$$

An interesting observation about such finite differences of a polynomial is that if the polynomial is of degree n (that is, n is the highest exponent of x with nonzero coefficient) then $\Delta^n y_i = k$, a constant. For example, consider the polynomial

$$y = 2x^2 - x + 1$$

The first and second differences are tabulated in Table 9-2, in which we

TABLE 9-2. DIFFERENCES OF A POLYNOMIAL OF DEGREE 2

x	y	Δy_i	$\Delta^2 y_i$
0	1		
		1	
1	2		4
		5	
2	7		4
		9	
3	16		4
		13	
4	29		4
		17	
5	46		

have written the differences *between* the corresponding lines of the table. To see that the $\Delta^2 y_i$ are constant, consider the general equation of degree 2:

$$y = ax^2 + bx + c$$

Then

$$\begin{aligned} y + \Delta y &= a(x + h)^2 + b(x + h) + c \\ &= ax^2 + (2ah + b)x + (ah^2 + bh + c) \end{aligned}$$

On subtracting the former equation from the latter we obtain

$$\Delta y = 2ahx + (ah^2 + bh)$$

Again,

$$\Delta y + \Delta(\Delta y) = 2ah(x + h) + (ah^2 + bh) = 2ahx + (3ah^2 + bh)$$

On subtracting, we find $\Delta(\Delta y) = \Delta^2 y = 2ah^2$, which is the desired constant. The same procedure for a general polynomial of the nth degree, namely,

$$y = ax^n + bx^{n-1} + \cdots + c$$

yields

$$\Delta^n y = a[n(n-1)(n-2)\cdots 1]h^n = ah^n n!$$

Thus, if the values of a polynomial of degree n are tabulated *at equal intervals, any* error could be detected by computing all the $\Delta^n y_i$ and checking for constancy.

Error Propagation of Differences. Let y_i ($i = 0, 1, 2, \ldots, 10$) be 11 correct values of the function $y(x)$, not necessarily a polynomial, for equal intervals of x. Suppose, however, that in the table of differences, Table 9-3, y_5 was in error and $y_5 + \epsilon$ had been recorded. The table shows how this error is propagated. Thus, for example,

$$\begin{aligned} y_5 + \epsilon - y_4 &= (y_5 - y_4) + \epsilon = \Delta y_5 + \epsilon \\ y_6 - (y_5 + \epsilon) &= (y_6 - y_5) - \epsilon = \Delta y_5 - \epsilon \\ \Delta y_4 + \epsilon - \Delta y_3 &= (\Delta y_4 - \Delta y_3) + \epsilon = \Delta^2 y_3 + \epsilon \\ (\Delta y_5 - \epsilon) - (\Delta y_4 + \epsilon) &= (\Delta y_5 - \Delta y_4) - 2\epsilon = \Delta^2 y_4 - 2\epsilon \end{aligned}$$

and so forth. Note that the coefficients of the ϵ's are just the *binomial coefficients with alternating signs* and that the *algebraic sum of the errors in each column is zero.* The *maximum error* in the differences is in the same horizontal line as the error in the function table. Thus, by taking a sufficient number of differences, an error in the table of a function can be identified. However, in practice it rarely pays to take more than the fourth differences, for beyond this the round-off errors themselves begin to accumulate.

TABLE 9-3. PROPAGATION OF ERRORS IN DIFFERENCES

y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$
y_0				
	Δy_0			
y_1		$\Delta^2 y_0$		
	Δy_1		$\Delta^3 y_0$	
y_2		$\Delta^2 y_1$		$\Delta^4 y_0$
	Δy_2		$\Delta^3 y_1$	
y_3		$\Delta^2 y_2$		$\Delta^4 y_1 + \epsilon$
	Δy_3		$\Delta^3 y_2 + \epsilon$	
y_4		$\Delta^2 y_3 + \epsilon$		$\Delta^4 y_2 - 4\epsilon$
	$\Delta y_4 + \epsilon$		$\Delta^3 y_3 - 3\epsilon$	
$y_5 + \epsilon$		$\Delta^2 y_4 - 2\epsilon$		$\Delta^4 y_3 + 6\epsilon$
	$\Delta y_5 - \epsilon$		$\Delta^3 y_4 + 3\epsilon$	
y_6		$\Delta^2 y_5 + \epsilon$		$\Delta^4 y_4 - 4\epsilon$
	Δy_6		$\Delta^3 y_5 - \epsilon$	
y_7		$\Delta^2 y_6$		$\Delta^4 y_5 + \epsilon$
	Δy_7		$\Delta^3 y_6$	
y_8		$\Delta^2 y_7$		$\Delta^4 y_6$
	Δy_8		$\Delta^3 y_7$	
y_9		$\Delta^2 y_8$		
	Δy_9			
y_{10}				

9-4. Function Evaluation: Interpolation

Polynomial Interpolation. Until now we have been considering functions that are memorized in the computer as subroutines. For instance, if $y = f(x)$, then for a given x_1 the computer will determine $f(x_1)$ by means of a program, which might be based on some expansion of $f(x)$. On the other hand, it is often impossible to write a program for a certain function: e.g., the function might be empirical data, or perhaps the subroutine might require too much time or too much memory space. In these and other instances it is often convenient to record values of $f(x)$ in the computer memory in the form of a function table which gives the $f(x_i)$ corresponding to a set of specific values of x_i. It appears at first glance that this method has great limitations, since only those values of $f(x)$ which correspond to a relatively few values of x_i are in the memory of the computer. However, by means of *interpolation*, values of $f(x)$ corresponding to an x not listed in the table can be approximated.

Suppose that it is desired to find the value of $f(x)$ for a value of x not listed in the table. It may happen that there are listed in the table some values of $f(x_i)$ for x_i near x. Then, in order to find the value of $f(x)$, it seems plausible to draw a smooth curve, $y = f^*(x)$, through all the points listed in the table and to say that $y = f^*(x)$ as read from this curve is an approximation to $y = f(x)$ (see Fig. 9-8). Such a process is called *interpolation* and is most often accomplished by means of a *polynomial approximation.*

If $n + 1$ points (x_0,y_0), (x_1,y_1), (x_2,y_2), . . . , (x_n,y_n) of $f(x)$ are listed in the table, then a *unique* polynomial of the nth degree, $f^*(x)$, can be derived that passes through *all* these $n + 1$ distinct points. For let

$$y = f^*(x) = A_0 + A_1x + A_2x^2 + \cdots + A_nx^n$$

If $f^*(x_0) = y_0$, $f^*(x_1) = y_1$, . . . , $f^*(x_n) = y_n$, then these are $n + 1$ equations which can be solved for A_0, A_1, . . . , A_n. Once these values have been obtained, we can find for any $x' \neq x_i$ the value of $f^*(x')$. The more points considered, i.e., the higher the degree of $f^*(x)$, the closer will be the value of $f^*(x')$ to $f(x')$. Let $f^*_{0,1}(x)$ represent the polynomial of first degree that goes through the points (x_0,y_0) and (x_1,y_1); let $f^*_{0,1,2}(x)$ represent the polynomial of second degree that goes through the points (x_0,y_0), (x_1,y_1), and (x_2,y_2); etc. Suppose that we want the interpolation $f^*(x')$ to within an accuracy of 0.001. To obtain this, we would calculate successively $f^*_{0,1}(x')$, $f^*_{0,1,2}(x')$, $f^*_{0,1,2,3}(x')$, etc., until

$$|f^*_{0,1,\ldots,r-1}(x') - f^*_{0,1,\ldots,r}(x')| < 0.001$$

and then we would take $f^*_{0,1,\ldots,r}(x')$ as our result.

Usually, the more known points used in an interpolation, the better becomes the accuracy of the interpolation. However, if the additional points are far from the point of interpolation, they may actually make the approximation worse. Similarly, with the same number of given points, the smaller the interval in which they are chosen, the more accurate the polynomial approximation in that interval. To illustrate

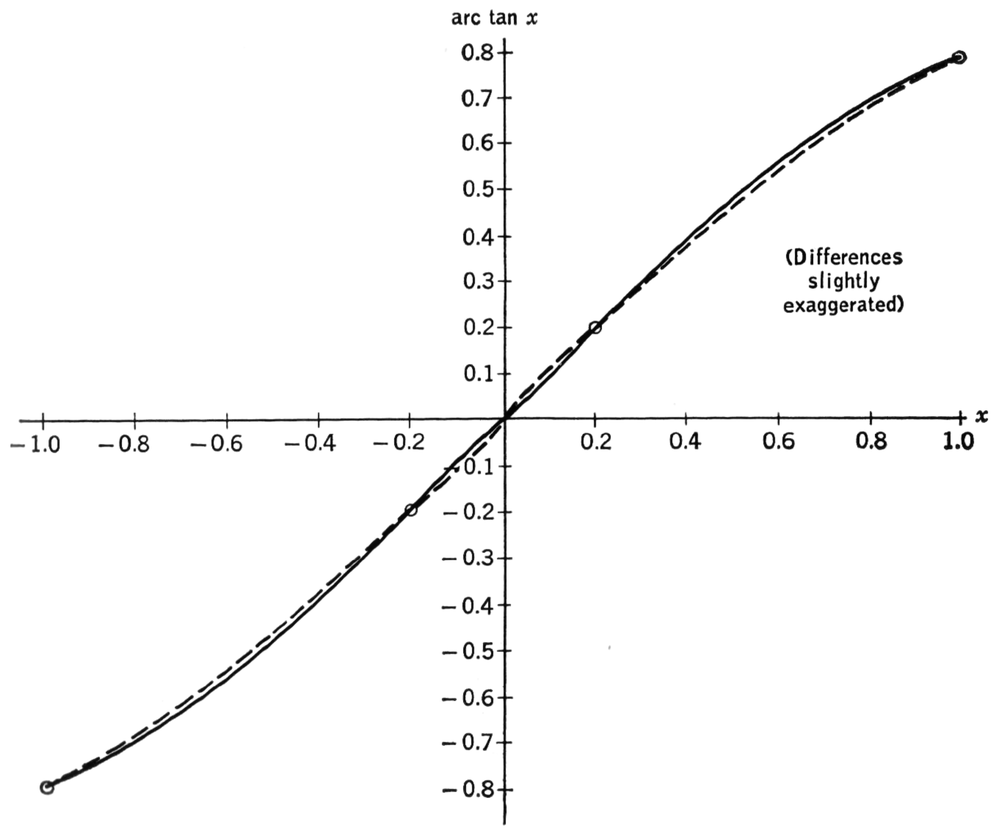


FIG. 9-8. A third-degree polynomial drawn through the four points $(-1, -0.785)$, $(-0.2, -0.197)$, $(+0.2, +0.197)$, $(+1, +0.785)$, as an approximation to arctan x, is shown by the solid line. Here $f^*(x) = \arctan^* x = 0.995x - 0.210x^3$. The dashed line is the actual arctan.

this latter point, in Fig. 9-9 we have drawn the *error curves* for two polynomial approximations to arctan x. Each error curve has x as abscissa and the difference between the actual value of arctan x and its polynomial approximation as the ordinate. The first approximation is based on the six points -1, -0.6, -0.2, $+0.2$, $+0.6$, and 1; the second approximation, on the six points 0, 0.2, 0.4, 0.6, 0.8, and 1. Figure 9-9 illustrates the error curves for $0 \leq x \leq 1$. Note that in this interval the maximum error in the former case is over forty times the maximum error in the latter case.

Obviously the sequence of calculations for a polynomial interpolation would be quite lengthy for each desired value if all the coefficients had to be calculated first, and then $f^*(x')$. The process can be shortened by calculating $f^*(x')$ directly, at essentially the same time as the coefficients are calculated. That is, the coefficients are never explicitly given, but

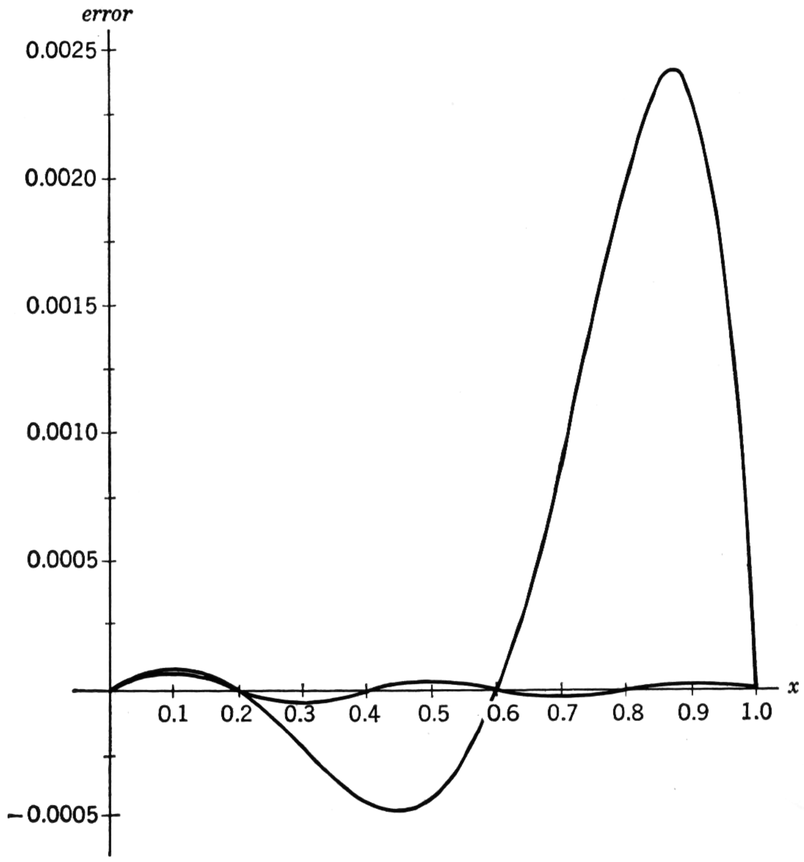


FIG. 9-9. Comparison of the error curves for the interpolation polynomial of arctan* x based on the six points -1, -0.6, -0.2, $+0.2$, $+0.6$, $+1$ (the curve with the big hump) and based on the six points 0, 0.2, 0.4, 0.6, 0.8, 1 (the curve that hugs the x axis). These error curves indicate just how much the interpolation polynomials differ from the true value of arctan x.

$f^*(x')$ results directly. It should be remembered, however, that the meaning of $f^*(x')$ is the value at x' of the approximating polynomial that goes through the specified points.

Aitken's Method. One method for shortening calculations, called Aitken's method, follows.† First note that

$$f^*_{0,1}(x) = \frac{\begin{vmatrix} y_0 & x_0 - x \\ y_1 & x_1 - x \end{vmatrix}}{x_1 - x_0} \qquad (a)$$

† See W. E. Milne, "Numerical Calculus," pp. 68–72, Princeton University Press, Princeton, N.J., 1949.

where $\begin{vmatrix} y_0 & x_0 - x \\ y_1 & x_1 - x \end{vmatrix}$ is the determinant with value $y_0(x_1 - x) - y_1(x_0 - x)$.

For

$$f^*_{0,1}(x_0) = \frac{\begin{vmatrix} y_0 & 0 \\ y_1 & x_1 - x_0 \end{vmatrix}}{x_1 - x_0} = \frac{y_0(x_1 - x_0)}{x_1 - x_0} = y_0$$

and

$$f^*_{0,1}(x_1) = \frac{\begin{vmatrix} y_0 & x_0 - x_1 \\ y_1 & 0 \end{vmatrix}}{x_1 - x_0} = y_1$$

Since $f^*_{0,1}(x)$ goes through the points (x_0,y_0) and (x_1,y_1) and is of the first degree, it is the required (unique) polynomial. Hence, for a given number x', to find $f^*_{0,1}(x')$ we just calculate Eq. (a) with this number x' substituted for the symbol x.

The polynomial that goes through (x_0,y_0), (x_1,y_1), and (x_2,y_2) is given by

$$f^*_{0,1,2}(x) = \frac{\begin{vmatrix} f^*_{0,1}(x) & x_1 - x \\ f^*_{0,2}(x) & x_2 - x \end{vmatrix}}{x_2 - x_1}$$

where $f^*_{0,2}(x)$ is the polynomial that goes through (x_0,y_0) and (x_2,y_2).

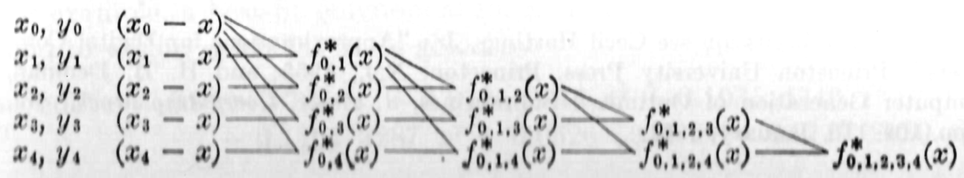
For $f^*_{0,1,2,3}(x)$ we have similarly

$$f^*_{0,1,2,3}(x) = \frac{\begin{vmatrix} f^*_{0,1,2}(x) & x_2 - x \\ f^*_{0,1,3}(x) & x_3 - x \end{vmatrix}}{x_3 - x_2}$$

The process can be extended in this way. For example,

$$f^*_{0,1,2,3,4,5,6}(x) = \frac{\begin{vmatrix} f^*_{0,1,2,3,4,5}(x) & x_5 - x \\ f^*_{0,1,2,3,4,6}(x) & x_6 - x \end{vmatrix}}{x_6 - x_5}$$

Hence the process can be systematized, with $f^*(x)$ being recomputed, including more and more points, until the desired accuracy is obtained (assuming that there are enough points in the table). The following scheme can be used for performing the calculations, the terms in each *row* being determined in order, until $f^*(x)$ at the end of a row is of the desired accuracy:



9-5. Function Evaluation: Best-fit Polynomial Approximation

Polynomial Approximations. As we have noted in Sec. 9-3, the evaluation of $f(x)$ by means of terms up to the x^n term of its power-series expansion can be considered as approximating $f(x)$ by a polynomial of degree n, where the coefficients are given by Taylor's formula. Similarly the method of interpolation can also be interpreted as the approximation of $f(x)$ by a polynomial, even though the polynomial may never be explicitly determined by the method given; the coefficients of the interpolation polynomial of degree n are determined from $n + 1$ known values of $f(x)$. For arctan x we have the power-series approximation

$$\arctan^* x = 1.000000x - 0.333333x^3 + 0.200000x^5$$

For the interpolation approximation (for the points -1, -0.6, -0.2, $+0.2$, $+0.6$, and 1 given in Sec. 9-4) we would have

$$\arctan^* x = 0.999105x - 0.306897x^3 + 0.093190x^5$$

In order to show how good these approximations are, in Fig. 9-10 have been drawn the error functions

$$\epsilon(x) = \text{approximation} - \text{function} = \arctan^* x - \arctan x$$

for the range $0 \leq x \leq 1$, for both cases.

Best-fit Polynomials. The question immediately arises: Does there exist another fifth-degree polynomial that is a "better fit" to arctan x for $0 \leq x \leq 1$? We say that $f_1^*(x)$ is a *better fit* to $f(x)$ than $f_2^*(x)$ for a given range of x if within that range $\max |\epsilon_1(x)| < \max |\epsilon_2(x)|$. In general the answer to the above question is "yes." For arctan x the best-fit polynomial for $0 \leq x \leq 1$ is given by

$$\arctan^* x = 0.995354x - 0.288679x^3 + 0.079331x^5$$

Figure 9-10 also shows $\epsilon(x)$ for this approximation. How one arrives at such best-fit approximations is beyond the scope of this book.†

The advantage of using a best-fit approximation over a Taylor-series approximation or interpolation is easily seen: fewer computations are required to obtain the same accuracy than for a Taylor series; and no tables need be memorized in the computer, as is necessary for interpolation. Hence time and space are saved in the computation. Although

† For such a discussion see Cecil Hastings, Jr., "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955; and H. H. Denman, Computer Generation of Optimized Subroutines, *J. Assoc. Computing Mach.*, vol. 8, pp. 104–116, January, 1961.

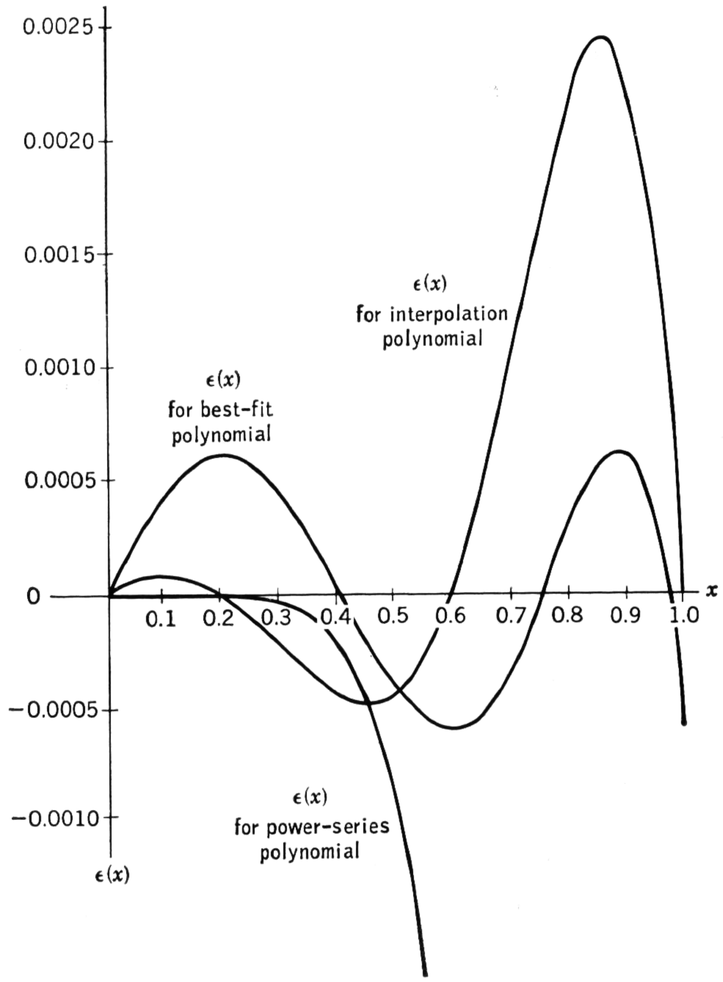


FIG. 9-10. A comparison, by means of error curves, of three fifth-degree polynomial approximations to arctan x for $0 \leq x \leq 1$.

1. Power-series approximation

$$\arctan^* x = 1.000000x - 0.333333x^3 + 0.200000x^5$$

2. Interpolation approximation on the points -1, -0.6, -0.2, $+0.2$, $+0.6$, and 1:

$$\arctan^* x = 0.999105x - 0.306897x^3 + 0.093190x^5$$

3. Best-fit approximation:

$$\arctan^* x = 0.995354x - 0.288679x^3 + 0.079331x^5$$

we have illustrated a best-fit polynomial that is good to within ± 0.0005, best-fit polynomials can be formed that will give any desired accuracy. For example, a best-fit polynomial for arctan x, for $0 \leq x \leq 1$, which is good to within $\pm 0.0000\ 01$, is given by

$$\arctan^* x = 0.9999\ 7726\ x - 0.3326\ 2347\ x^3 + 0.1935\ 4346\ x^5 - 0.1164\ 3287\ x^7 + 0.0526\ 5332\ x^9 - 0.0117\ 2120\ x^{11}$$

Best-fit approximations need not be of the forms shown for arctan x but should be of some relatively simple algebraic form. For example, for $0 \leq x \leq \infty$, a best-fit approximation for

$$f(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

is given by

$$f^*(x) = 1 - \frac{1}{(1 - a_1x + a_2x^2 + a_3x^3 + a_4x^4)^4}$$

where $a_1 = 0.278393$, $a_2 = 0.230389$, $a_3 = 0.000972$, and $a_4 = 0.078108$.

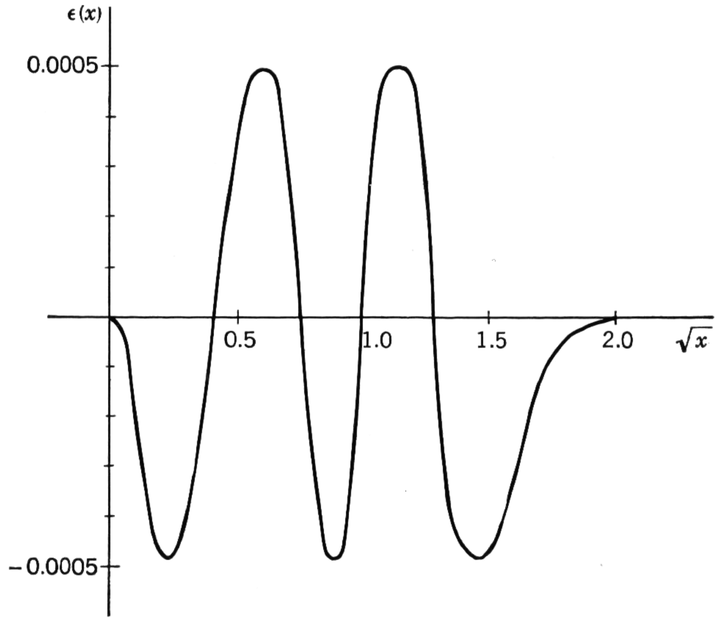


FIG. 9-11. Error curve for $f(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$.

In this case $\epsilon(x)$ is given in Fig. 9-11, plotted against $\sqrt{x}$. This $f^*(x)$ is easily coded for a computer.

9-6. Integration

By Definition. In this section we shall see how a computer can integrate. In his previous studies the student has been taught to integrate by means of the manipulation of abstract symbols. However, a little reflection will show that it is really numbers being handled, these numbers being represented for convenience by symbols. Of course, a computer cannot directly manipulate symbols in the ordinary sense, but it can handle numbers. As we shall see, the simplest method of handling

numbers to obtain an integral is to follow the very definition of the meaning of integration.

Recall now the definition of integration (see Fig. 9-12). The integral of $f(x)$ from $x = 0$ to $x = p$, written $\int_0^p f(x)\, dx$, is the area between the curve $y = f(x)$ and the x axis, bounded by $x = 0$ and $x = p$. It is found by considering the rectangles formed by partitioning the x axis between 0 and p into n intervals each of the same length h, where $h = (p - 0)/n$. If there are n intervals each of length h, the area of the ith rectangle will be hy_i and the sum of the areas of all the rectangles will be

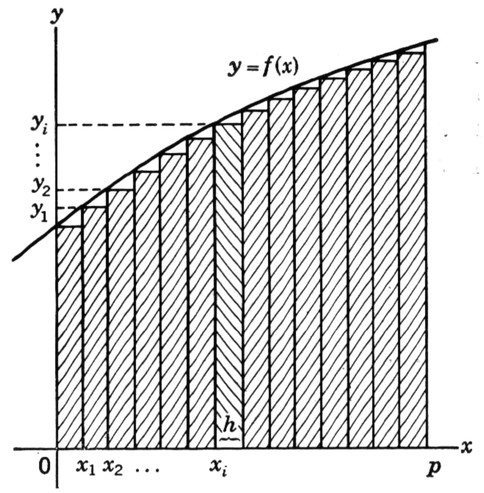


FIG. 9-12. Numerical integration.

$$A = \sum_{i=1}^{n} hy_i$$

In another partition that has more intervals the length of each will be less. Hence, as the number of intervals becomes infinite, the length of each approaches 0. The integral $\int_0^p y\, dx$ is defined as the limit of the sum of the areas of the rectangles as the number of intervals increases and the length of each approaches zero,

$$\int_0^p y\, dx = \lim_{\substack{n \to \infty \\ h \to 0}} \sum_{i=1}^{n} hy_i$$

To integrate on a computer, we could just form $\sum_{i=1}^{n} hy_i$. Then we might take a smaller interval, say $h/2$, and form $\sum_{i=1}^{2n} (h/2)y_i$. If

$$\left| \sum_{i=1}^{2n} \frac{h}{2} y_i - \sum_{i=1}^{n} hy_i \right|$$

is less than the allowable error desired for this integration, then the integration is completed. If not, we could try intervals of $h/4$, and so on. This process naturally assumes that we have a subroutine to evaluate $y_i = f(x_i)$ for each x_i under consideration.

By the Trapezoid Rule. Although the above method is a good way for evaluating an integral, there are methods that may require less computation. Suppose that we considered the trapezoids formed by adding little triangles to the rectangles, as shown in Fig. 9-13. Here the area of the trapezoid is $h[y_0 + \frac{1}{2}(y_1 - y_0)] = (h/2)(y_0 + y_1)$. For n trapezoids we have $A = (h/2)(y_0 + 2y_1 + 2y_2 + \cdots + y_n)$. Certainly this will be a better initial approximation to the area under the curve than was $A = h(y_1 + y_2 + \cdots + y_n)$. Hence, using this method, we shall not need to go to as small intervals to obtain the desired accuracy as we did using the previous, cruder method. The method given in this paragraph is called the *trapezoid rule.*

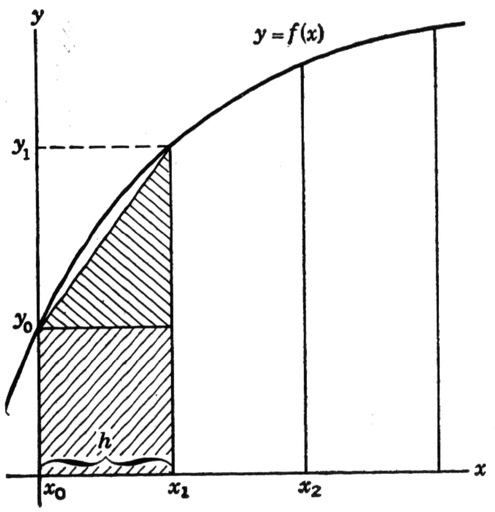


FIG. 9-13. Trapezoidal approximation to integration.

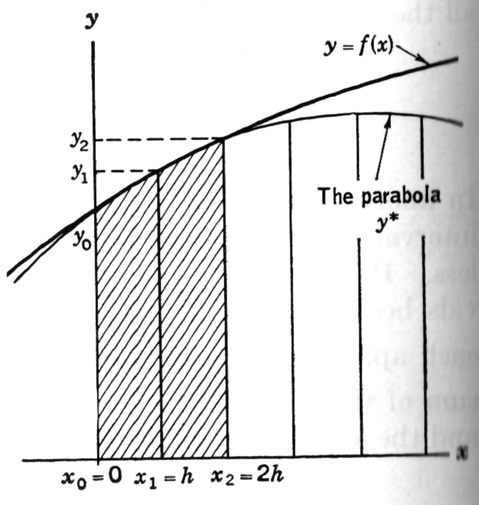


FIG. 9-14. Parabolic approximation to integration.

By Polynomial Approximations. An even better initial approximation can be obtained by considering three points at a time instead of two. Suppose that we consider the area under the parabola $y^* = a + bx + cx^2$ that passes through three points, as shown in Fig. 9-14. Then

$$A_{0,2} = \int_0^{2h} (a + bx + cx^2)\,dx = ax + \frac{bx^2}{2} + \frac{cx^3}{3}\Big]_0^{2h}$$
$$= 2h \cdot a + 2h^2 \cdot b + \tfrac{8}{3}h^3 \cdot c$$

To determine the coefficients a, b, and c, we note that

If $x = 0$, $\quad a + 0 \cdot b + 0 \cdot c = y_0$
If $x = h$, $\quad a + h \cdot b + h^2 \cdot c = y_1$
If $x = 2h$, $\quad a + 2h \cdot b + 4h^2 \cdot c = y_2$

These, when solved for a, b, and c, yield

$$a = y_0 \qquad b = \frac{-y_2 + 4y_1 - 3y_0}{2h} \qquad c = \frac{y_2 - 2y_1 + y_0}{2h^2}$$

whence

$$A_{0,2} = \frac{h}{3}(y_0 + 4y_1 + y_2)$$

Similarly $A_{2,4} = (h/3)(y_2 + 4y_3 + y_4)$, and so forth, whence, if n is even, we have

$$A = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + 4y_5 + \cdots + 4y_{n-1} + y_n)$$

This is known as *Simpson's rule.*

Similarly we could try to make even better initial approximations to the area under the curve. Once the method of approximation is decided upon, more accurate results can be obtained only by increasing n, the number of intervals—as can be seen from the original definition of the integral.

We have been assuming until now that the function $y(x)$ can be evaluated at any point x by a subroutine. Suppose, on the other hand, that the function $f(x)$ were given by a table of points, spaced at equal intervals x_i. Then the above procedures would of course still hold. But these methods will evaluate integrals only between the points that are given in the table, and not other points. However, one might choose simply to integrate the approximating polynomial (determined as in the previous sections) and then to evaluate this function between any two points, whether they are or are not given in the table.

9-7. Differentiation

By Definition. The derivative of $y = f(x)$ with respect to x, taken at $x = x_0$, is defined by

$$\left(\frac{dy}{dx}\right)_{x=x_0} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

and represents the slope of the tangent drawn to the curve $y = f(x)$ at x_0. Hence, if $f(x)$ is given at $x = x_0$ and $x = x_1$, an approximation of the derivative might be

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

Unfortunately, with computers it is not always feasible to take x_1 successively closer to x_0 and evaluate the fraction each time, for $x_1 - x_0$ becomes very small, as does $f(x_1) - f(x_0)$, and often for a very small $x_1 - x_0$ the loss of significant figures throws the calculation far off.

By Polynomial Approximation. Suppose that $f(x)$ is given at $x_0 = 0$, $x_1 = h$, and $x_2 = 2h$. Analogous to the process described for integration, we might determine as an approximation the derivative of the parabola that goes through these three points. Let $y = a + bx + cx^2$, where a, b, and c are as in Sec. 9-6. Hence

$$\frac{dy}{dx} = b + 2cx = \frac{-y_2 + 4y_1 - 3y_0}{2h} + \frac{y_2 - 2y_1 + y_0}{h^2}\,x$$

Then for the derivatives taken at $x = x_0$, $x = x_1$, and $x = x_2$, respectively, we have

$$\left(\frac{dy}{dx}\right)_{\substack{x=0\\y=y_0}} = \frac{1}{2h}(-3y_0 + 4y_1 - y_2)$$

$$\left(\frac{dy}{dx}\right)_{\substack{x=h\\y=y_1}} = \frac{1}{2h}(-y_0 + y_2)$$

$$\left(\frac{dy}{dx}\right)_{\substack{x=2h\\y=y_2}} = \frac{1}{2h}(y_0 - 4y_1 + 3y_2)$$

The formulas are equally valid for any three points (x_0,y_0), (x_1,y_1), and (x_2,y_2) equally spaced on the x axis, since translating the curve so that

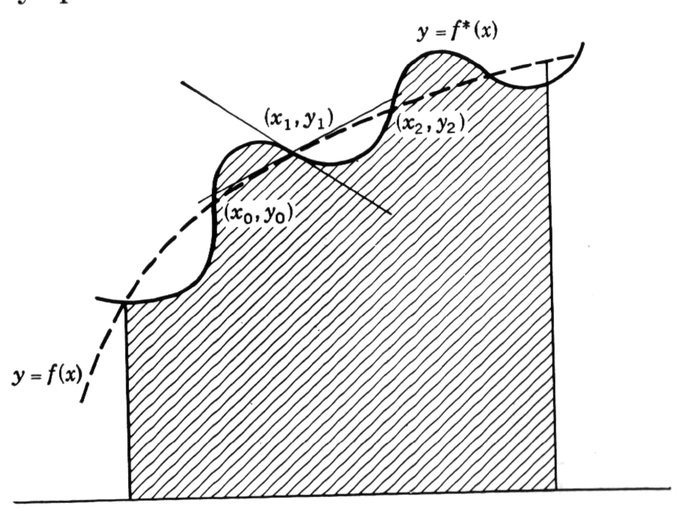


FIG. 9-15. Approximating the derivative from an oscillating polynomial.

$x_0 = 0$ does not affect its slope.† If the function is given by means of a table, then these rules can be used for evaluating derivatives at points given in the table.

Of course there are always the approximating polynomials of $f(x)$, which can easily be differentiated, the resulting derivative being evaluated at the requisite points.

† See, for example, Milne, *op. cit.*, p. 96.

However, numerical differentiation at best is often quite inaccurate. For these methods must be based on approximating polynomials that depend on values of the function at a few specific points. There is no practical way for the computer to tell how the approximating polynomial oscillates around the given function. Hence the slope of the polynomial at a point may be entirely different from the slope of the function, even though the curves lie very close, or even touch (see Fig. 9-15). On the other hand, the value of the area under the curve is seldom so sensitive to these oscillations.

9-8. Undetermined Coefficients

The Method. As was shown in the previous two sections, formulas for the integral or derivative of a function can be obtained by integrating or differentiating the polynomial approximation functions that pass through some given points. However, in those sections formulas were specifically derived only where three points were fitted. Formulas based on more points can be derived by the method of undetermined coefficients,† which follows. We shall develop the method by means of illustrations.

Suppose that we wish to rederive the formula of Sec. 9-6, Simpson's rule, and find an approximation of $\int_{x_0}^{x_2} y\,dx$ based on the polynomial $y = f^*(x)$ of the second degree passing through the three points (x_0,y_0), (x_1,y_1), (x_2,y_2) with equally spaced abscissas. Since the value of an integral does not change if it is translated along the x axis, it will prove helpful to take advantage of symmetry, by letting the x coordinates of these points be $x_0 = -h$, $x_1 = 0$, $x_2 = +h$. Now let

$$A_0y_0 + A_1y_1 + A_2y_2 = \int_{x_0}^{x_2} y\,dx$$

where A_0, A_1, and A_2 are the as yet undetermined coefficients. This formula must be *exact* (i.e., not an approximation) for $y = f(x)$, when $f(x)$ is any *polynomial of degree 2 or less,* because there is a uniquely determined polynomial of degree not more than n that passes through $n + 1$ points. Hence this formula must be true for the functions $y = 1$, $y = x$, and $y = x^2$. Substituting these three functions successively in the equation, we find

$$\begin{aligned} A_0 + \quad A_1 + \quad A_2 &= 2h \\ -hA_0 + 0 \cdot A_1 + \; hA_2 &= 0 \\ h^2A_0 + 0 \cdot A_1 + h^2A_2 &= \tfrac{2}{3}h^3 \end{aligned}$$

† See *ibid.*, p. 104.

Solving for A_0, A_1, and A_2, we find

$$A_0 = A_2 = \tfrac{1}{3}h \qquad \text{and} \qquad A_1 = \tfrac{4}{3}h$$

Hence

$$\int_{x_0}^{x_2} y\,dx = \frac{h}{3}(y_0 + 4y_1 + y_2)$$

as desired.

Suppose that we now desire to find a formula for $(dy/dx)_{x=x_1}$ based on four points with equally spaced abscissas, (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3). Let $x_0 = -h$, $x_1 = 0$, $x_2 = h$, $x_3 = 2h$, and

$$B_0y_0 + B_1y_1 + B_2y_2 + B_3y_3 = \left(\frac{dy}{dx}\right)_{x=x_1}$$

where B_0, B_1, B_2, and B_3 are the undetermined coefficients. This formula must be *exact* when y is any polynomial of degree not more than 3, and hence for the polynomials $y = 1$, $y = x$, $y = x^2$, and $y = x^3$. Substituting these successively into the equation, we find

$$\begin{aligned} B_0 + \quad B_1 + \quad B_2 + \quad B_3 &= 0 \\ -hB_0 + 0 \cdot B_1 + hB_2 + 2hB_3 &= 1 \\ h^2B_0 + 0 \cdot B_1 + h^2B_2 + 4h^2B_3 &= 0 \\ -h^3B_0 + 0 \cdot B_1 + h^3B_2 + 8h^3B_3 &= 0 \end{aligned}$$

These, when solved for B_0, B_1, B_2, and B_3, give

$$B_0 = -\frac{1}{3h} \qquad B_1 = -\frac{1}{2h} \qquad B_2 = \frac{1}{h} \qquad \text{and} \qquad B_3 = -\frac{1}{6h}$$

whence

$$\left(\frac{dy}{dx}\right)_{x=x_1} = \frac{1}{6h}(-2y_0 - 3y_1 + 6y_2 - y_3)$$

9-9. Differential Equations

Meaning of Solution. A differential equation describes how a system goes from a known point to the next infinitesimally close point. For instance, the well-known equation $dv/dt = F/m$ describes the relationship of the velocity v of the mass m to the time t, when a force F is applied. Given a point (t_0,v_0), the equation says that at an infinitesimally short time later, say at t_1, the velocity will be $v_1 = (F/m)(t_1 - t_0) + v_0$. Such equations arise because in nature relationships between physical quantities often reduce to linear functions at the infinitesimal level.

Consider, for example, a differential equation of the first order in the form

$$\frac{dy}{dx} = f(x,y)$$

The solution is to be determined as a function of x only, that is, $y = g(x)$. Of course $y = g(x)$ can be represented as a curve, and the differential

equation tells us how to draw this curve. One point on the curve, say (x_0,y_0), must be known or given. Then the differential equation says that

$$\left(\frac{dy}{dx}\right)_{x=x_0} = f(x_0,y_0)$$

and hence we know the slope at (x_0,y_0). A linear approximation to the next point might be obtained (see Fig. 9-16) by continuing along

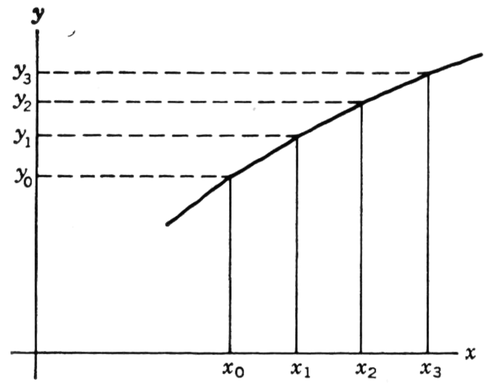


FIG. 9-16. Linear approximation from point to point along a curve.

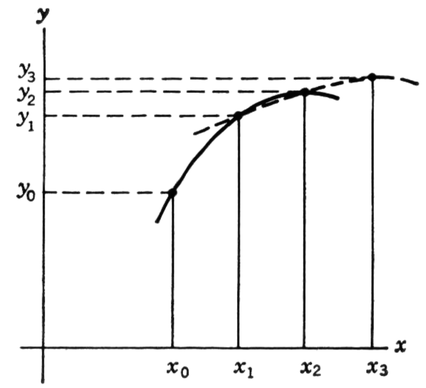


FIG. 9-17. Polynomial approximation from point to point along a curve.

the tangent line, i.e., by choosing an x_1 and determining y_1 from

$$y_1 = y_0 + (x_1 - x_0)\left(\frac{dy}{dx}\right)_{x=x_0}$$

Using (x_1y_1), we find that $(dy/dx)_{x=x_1} = f(x_1,y_1)$ and then, choosing an x_2, determine y_2 from

$$y_2 = y_1 + (x_2 - x_1)\left(\frac{dy}{dx}\right)_{x=x_1}$$

and in this manner (x_3,y_3), . . . are determined and the function $y = g(x)$ is approximated. Of course this is just an approximation, even if we make $x_1 - x_0$, $x_2 - x_1$, $x_3 - x_2$, . . . small, for the differential equation taken literally implies that $x_1 - x_0$, $x_2 - x_1$, . . . should be infinitesimal. What we are essentially saying by this method is that a straight line through (x_0,y_0) is a good approximation to $g(x)$ near (x_0,y_0) and therefore the next point (x_1,y_1) ought to lie very close to it.

By Polynomial Approximation. Consider again a first-order differential equation $dy/dx = f(x,y)$. A better approximation to $y = g(x)$ might be obtained if we started with two points, (x_0,y_0) and (x_1,y_1). Then we could use a parabola as the approximating curve of $g(x)$ near (x_0,y_0) and (x_1,y_1), to determine (x_2,y_2) (see Fig. 9-17). Take, for example, the

second formula for the parabolic approximation of Sec. 9-7,

$$\left(\frac{dy}{dx}\right)_{x=x_1} = \frac{1}{2h}(-y_0 + y_2)$$

From this

$$y_2 = y_0 + 2h\left(\frac{dy}{dx}\right)_{x=x_1}$$

Hence if we take $x_1 = x_0 + h$, $x_2 = x_0 + 2h$, etc., and if we know (x_0,y_0) and (x_1,y_1), then we can determine $(dy/dx)_{x=x_1} = f(x_1,y_1)$, and from this y_2. Our equation from Sec. 9-7 can be written in general as

$$y_{n+1} = y_{n-1} + 2h\left(\frac{dy}{dx}\right)_{x=x_n}$$

Thence we find y_3 for $x_3 = x_0 + 3h$, using $(dy/dx)_{x=2} = f(x_2,y_2)$, and so forth. This process is clear, provided that initially we know (x_1,y_1) as well as (x_0,y_0). Given (x_0,y_0), we can find (x_1,y_1) by forming a Taylor-series expansion around (x_0,y_0). Since we already know $dy/dx = f(x,y)$, then

$$\frac{d^2y}{dx^2} = \frac{d}{dx}f(x,y) \qquad \frac{d^3y}{dx^3} = \frac{d^2}{dx^2}f(x,y) \qquad \text{etc.}$$

for the coefficients, and hence (x_1,y_1) is easily found,

$$y_1 = y_0 + (x_1 - x_0)\left(\frac{dy}{dx}\right)_{x=x_0} + \frac{(x_1 - x_0)^2}{2!}\left(\frac{d^2y}{dx^2}\right)_{x=x_0} + \frac{(x_1 - x_0)^3}{3!}\left(\frac{d^3y}{dx^3}\right)_{x=x_0} + \cdots$$

where the first few terms are usually sufficient.

Similarly, higher-degree approximation polynomials can be used to estimate successive points. For example, a cubic might be used that is based on

$$(x_0,y_0) \qquad (x_1,y_1) \qquad \left(\frac{dy}{dx}\right)_{x=x_0} \qquad \text{and} \qquad \left(\frac{dy}{dx}\right)_{x=x_1}$$

Such formulas can be derived by means of the method of undetermined coefficients.

Example. As an example, consider the differential equation

$$\frac{dy}{dx} = 1 + y^2$$

Suppose that, at $x_0 = 0$, $y_0 = 0$. Choose $x_1 = 0.1$, $x_2 = 0.2$, $x_3 = 0.3$, etc. Then, by our first, linear method,

$$y_1 = y_0 + (x_1 - x_0)\left(\frac{dy}{dx}\right)_{x=x_0} = y_0 + (x_1 - x_0)(1 + y_0^2)$$
$$= 0 + (0.1 - 0)(1 + 0) = 0.1$$

$$y_2 = y_1 + (x_2 - x_1)\left(\frac{dy}{dx}\right)_{x=x_1} = y_1 + (x_2 - x_1)(1 + y_1^2)$$
$$= 0.1 + 0.1 \times 1.01 = 0.201$$

$$y_3 = y_2 + (x_3 - x_2)\left(\frac{dy}{dx}\right)_{x=x_2} = y_2 + (x_3 - x_2)(1 + y_2^2) = 0.3050$$

and so forth.

Next consider the quadratic method. We must first find y_1 for $x_1 = 0.1$. Taking the first three terms of the Taylor expansion,

$$y_1 = y_0 + (x_1 - x_0)(1 + y_0^2) + \frac{(x_1 - x_0)^2}{2!}2y_0(1 + y_0^2) + \frac{(x_1 - x_0)^3}{3!}[2(1 + y_0^2)^2 + 4y_0^2(1 + y_0^2)]$$
$$= 0 + 0.1 + 0 + 0.0003$$
$$= 0.1003$$

Then

$$y_2 = 0.2\left(\frac{dy}{dx}\right)_{x=x_1} + y_0 = 0.2 \times (1 + y_1^2) + y_0$$
$$= 0.2 \times 1.0101 + 0 = 0.2020$$

$$y_3 = 0.2\left(\frac{dy}{dx}\right)_{x=x_2} + y_1 = 0.2 \times (1 + y_2^2) + y_1$$
$$= 0.2 \times 1.0408 + 0.1003 = 0.3085$$

and so forth. The exact solution to $dy/dx = 1 + y^2$ is $y = \tan x$. With greater accuracy, we have $y_0 = \tan 0 = 0$, $y_1 = \tan 0.1 = 0.1021$, $y_2 = \tan 0.2 = 0.2035$, and $y_3 = \tan 0.3 = 0.312$.

Since in the numerical solution of a differential equation each point depends on previous points, errors are cumulative. Hence the only way to get a better solution is to take the interval h smaller for a particular method of approximation.

By Runge's Method. There are formulas for the integration of differential equations other than those which can be derived directly by the method of undetermined coefficients. One well-known example is Runge's method. This has the advantages that it has very small error and that it requires no special methods to start the solution. It is, however, based on Simpson's rule given in Sec. 9-6 above, which for $y = F(x)$ can be written

$$A_{0,h}(F) = \int_{x_0}^{x_0+h} F(x)\,dx = \frac{h}{6}\left[F(x_0) + 4F\left(x_0 + \frac{h}{2}\right) + F(x_0 + h)\right]$$

If we have $y'(x) = dy/dx = f(x,y)$, then

$$y(x_0 + h) = y(x_0) + \Delta y = y(x_0) + \int_{x_0}^{x_0+h} y'(x)\,dx = y(x_0) + A_{0,h}(y')$$

That is, $\Delta y = A_{0,h}(y')$; or, applying Simpson's rule,

$$\Delta y = \frac{h}{6}(y'_0 + 4y'_1 + y'_2)$$

Clearly $y'_0 = f(x_0,y_0)$ can be computed. The approximation

$$y'_1 \cong f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}y'_0\right)$$

is reasonable since it involves only $h/2$. However,

$$f(x_0 + h, y_0 + y'_0 h) = m_2$$

is too crude an approximation for y'_2, and hence Runge's method uses $y'_2 \cong f(x_0 + h, y_0 + m_2 h)$. Summarizing, then, the method uses these formulas:

$$m_0 = f(x_0,y_0)$$
$$m_1 = f\left(x_0 + \frac{h}{2}, y_0 + m_0\frac{h}{2}\right)$$
$$m_2 = f(x_0 + h, y_0 + m_0 h)$$
$$m_3 = f(x_0 + h, y_0 + m_2 h)$$

from which
$$\Delta y = \frac{h}{6}(m_0 + 4m_1 + m_3)$$

Figure 9-18 gives the geometric interpretation of the different slopes m_0, m_1, m_2, and m_3 used.

9-10. Partial Differential Equations

Obtaining numerical solutions to partial differential equations can be a difficult task. Ideally the nature and properties of the solutions should be known beforehand. This knowledge can be obtained through mathematical analytic investigations and also from the behavior of the physical situation that the equation describes. The uniqueness of the solution must be assured, as well as the continuity of the solution in the region under consideration. There are many methods for solving partial differential equations numerically, some of which depend strongly on the nature of the particular equation under consideration. In this short section we can do no more than discuss some of the concepts upon which these numerical methods are often based, a more advanced treatment being beyond the scope of this book.

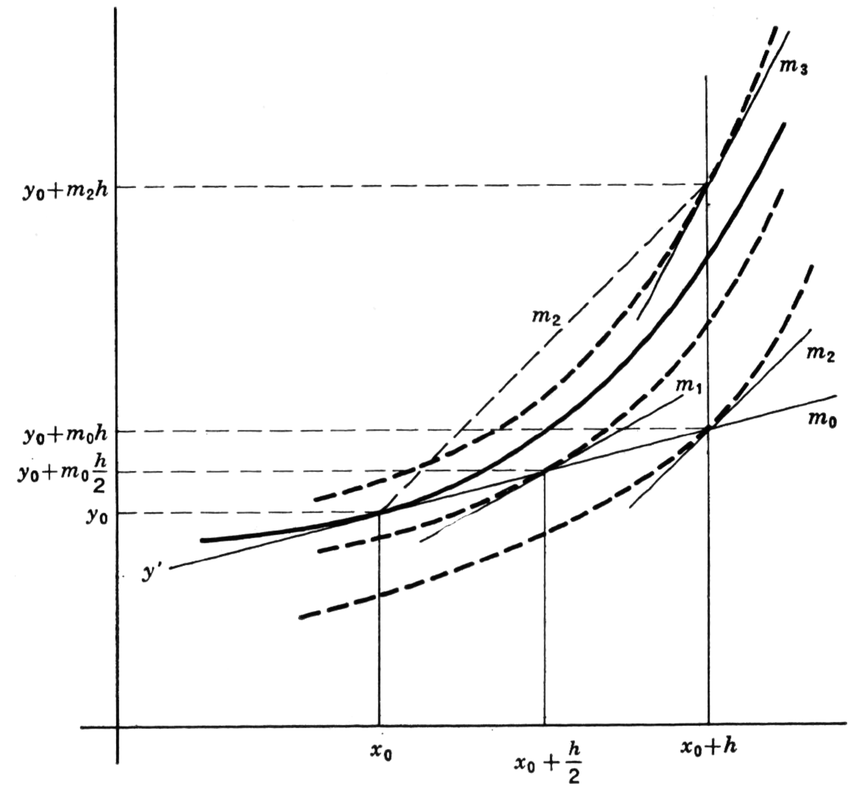


FIG. 9-18. Illustration of slopes m_0, m_1, m_2, and m_3 used in Runge's method.

Difference Quotients. A difference quotient is the finite approximation to a derivative; it is the quotient whose limiting value is the derivative. For example, if $f = f(x)$,

$$f_x = \frac{f(x + h) - f(x)}{h}$$

is the difference quotient corresponding to df/dx (see Fig. 9-19*a*). Similarly the "backward" difference quotient

$$f_{\bar{x}} = \frac{f(x) - f(x - h)}{h}$$

could be formed. For functions of two variables, difference quotients corresponding to partial derivatives can be formed. For example, if $u = u(x,y)$, then corresponding to $\partial u/\partial x$ we have the difference quotients

$$u_x = \frac{u(x + h, y) - u(x,y)}{h}$$
$$u_{\bar{x}} = \frac{u(x,y) - u(x - h, y)}{h}$$

and corresponding to $\partial u/\partial y$ we can form

$$u_y = \frac{u(x, y + h) - u(x,y)}{h}$$

$$u_{\bar{y}} = \frac{u(x,y) - u(x, y - h)}{h}$$

These can be visualized in Fig. 9-19*b*. Second difference quotients,

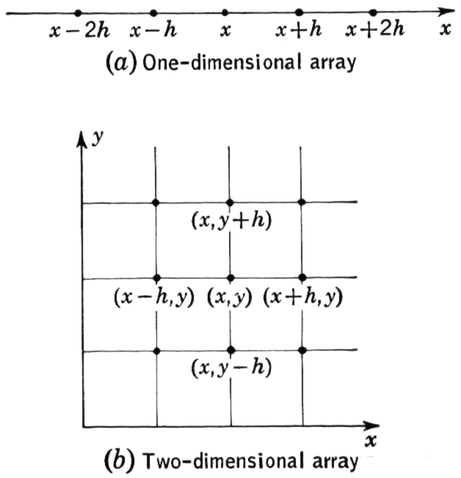


FIG. 9-19. Lattice arrays for visualizing difference quotients.

corresponding to second partial derivatives, are difference quotients of the first difference quotients. Thus we have

$$u_{\bar{x}x} = \frac{u_x - u_{\bar{x}}}{h} = \frac{u(x + h, y) - 2u(x,y) + u(x - h, y)}{h^2}$$

$$u_{\bar{y}y} = \frac{u_y - u_{\bar{y}}}{h} = \frac{u(x, y + h) - 2u(x,y) + u(x, y - h)}{h^2}$$

Analogous difference quotients can be written for functions of more than two variables and visualized in lattice arrays of more than two dimensions.

The solution to partial differential equations is cast in the form of operations with the *difference equations* formed by replacing the partial derivatives by their corresponding difference quotients. For example, consider the equation (known as Laplace's equation)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{1}$$

The corresponding difference equation becomes

$$\frac{u(x + h, y) - 2u(x,y) + u(x - h, y)}{h^2} + \frac{u(x, y + h) - 2u(x,y) + u(x, y - h)}{h^2} = 0 \tag{2}$$

If we solve for $u(x,y)$ the "meaning" of the equation becomes apparent, for

$$u(x,y) = \tfrac{1}{4}[u(x + h, y) + u(x - h, y) + u(x, y + h) + u(x, y - h)] \tag{3}$$

Thus the equation "says" that the value of u at any point is the average of the values of u at the four adjacent lattice-array points. As a physical example of such a situation, consider a square of metal as in Fig. 9-20. Suppose that the temperatures u at points along the edge of the square are fixed and that the temperature of the rest of the square has become stationary, i.e., is no longer changing with the time; then Laplace's equation (1) holds. That is, according to (3), the temperature at any point in the square is nearly the average of the temperatures at the four

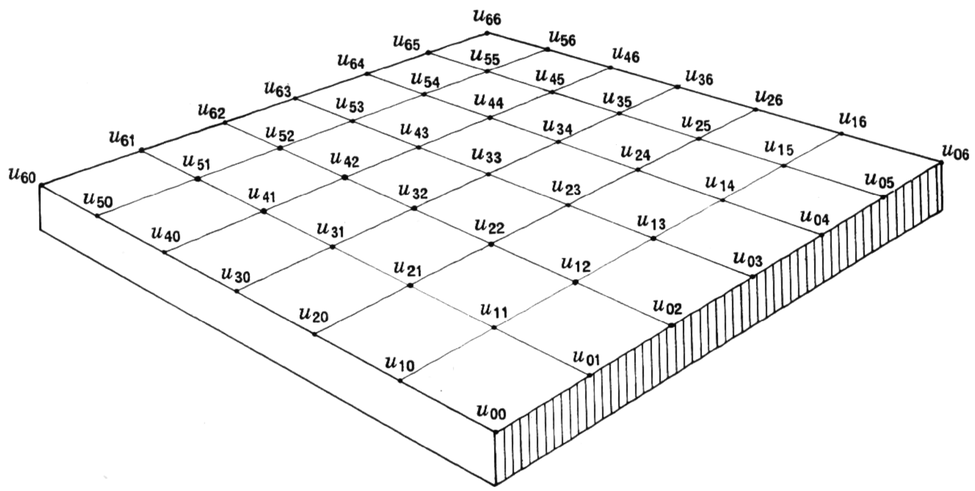


FIG. 9-20. Temperatures on a metal square to illustrate Laplace's equation.

adjacent points—this last remark holding only when h is sufficiently small, of course!

Thus the solution to (1) is found by determining the values of the function u at all lattice points so that (3) is satisfied. Then a smaller interval h is chosen, and (3) is satisfied again for this finer array of lattice points, and so forth, until the values at common points determined in two successive steps do not change appreciably. Summarizing, the numerical solution to the partial differential equation is the array of final values of the function at the lattice points as determined by both the given boundary values and the equation itself. For example, in Fig. 9-21 the vertical axis represents the temperature value of our square as computed for the boundary temperature values indicated by the dashed curves. Clearly, therefore, the values comprising the solution *need be no more accurate than the given values at the boundary.*

By Iteration of Difference Equations. The method of solution of partial differential equations by iteration is best described by means of an example. We shall therefore consider the Laplace equation (1) with the boundary values of Fig. 9-21. The first step is to start with a crude, or coarse, lattice and make a guess at the solution, i.e., at the values of the

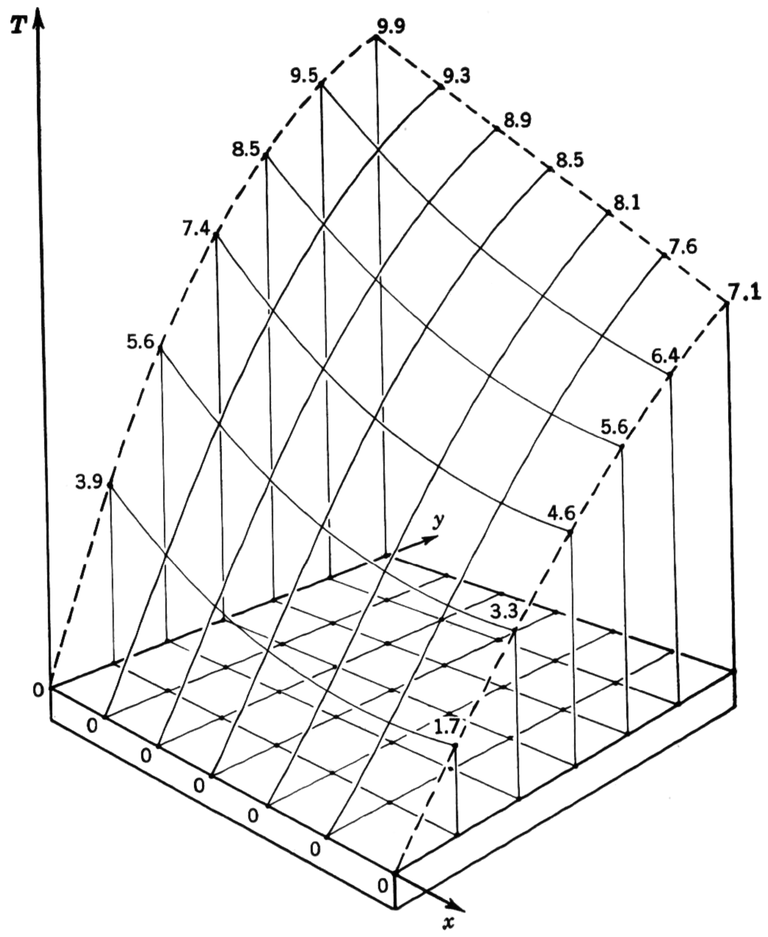


FIG. 9-21. Solution to Laplace's equation for the temperature example, with boundary values as shown.

function at these lattice points. For example, Fig. 9-22 illustrates a coarse lattice with appropriate guesses (top numbers in each list) at the lattice points. In this case it is reasonable to take as an initial guess the weighted average of the four associated boundary points as the value of an interior point. For instance, for u_{11} we formed

$$u_{11} = \frac{2u_{10} + 2u_{01} + u_{13} + u_{3}}{6} = 3.4$$

and so forth. Of course in a realistic situation all auxiliary information would be brought to bear on the initial guess and if possible a much finer lattice would be chosen. With the initial guess completed, the difference equation is utilized to successively and systematically adjust the values at each point by utilizing the values of the adjacent points. Each time a computation is made, the most recent values of the adjacent points are used. For our example we use Eq. (3) and determine successively u_{11}, u_{12}, u_{21}, and u_{22}. Thus

$$u_{11} = \tfrac{1}{4}(u_{12} + u_{10} + u_{21} + u_{01}) = 3.3$$

Next u_{12} is computed from

$$u_{12} = \tfrac{1}{4}(u_{13} + u_{11} + u_{22} + u_{01}) = 5.9$$

where the most recent value of u_{11} enters the calculation, and so forth. All the points of the lattice are successively recomputed over and over again until they no longer change.

u_{30}	u_{31}	u_{32}	u_{33}
0	5.6	8.5	9.9
u_{20}	u_{21}	u_{22}	u_{23}
0	3.9	6.7	8.9
u_{10}	u_{11}	u_{12}	u_{13}
0	3.4 3.3	6.0 5.9	8.1
u_{00}	u_{01}	u_{02}	u_{03}
0	3.3	5.6	7.1

FIG. 9-22. Initial values and first, coarse-lattice computations.

The next step is to increase the number of points on the lattice, say by halving the interval h (see Fig. 9-23). Since new points appear, initial guesses must again be made. Here interpolations can be made for new points that lie between two old points; the other new points might be given the value of the average of adjacent points. We choose a simple linear interpolation and average for the illustration. The initial guesses having been made, the iteration proceeds row by row for all points, the entire process being repeated until the values of all points cease changing. A finer lattice can then be used, and so forth. The resulting function $u(x,h)$ has been shown in three-dimensional form in Fig. 9-21.

By Relaxation of Difference Equations. For each fineness of array lattice points considered in the numerical processes, our aim is to determine the values of u_{ij} so that the difference equation (associated with the partial differential equation) is satisfied at every lattice point. The procedure is first to guess at initial values and then to keep systematically adjusting these until the equation holds at every point; then a finer array of lattice points is chosen and the procedure repeated. In the previous paragraph we considered an iterative method for adjusting the u_{ij} values at any array stage; here we shall consider another method for performing the adjustment of u_{ij} values, called the *relaxation* method for satisfying the difference equation. Again this method is best described by means of an example. As before let us choose Laplace's equation with difference equation (3) for the illustration.

The method of relaxation is concerned with the so-called *residual* R_{ij} at each point, which is the discrepancy in the value of u at the point that accounts for the difference equation *not* holding.

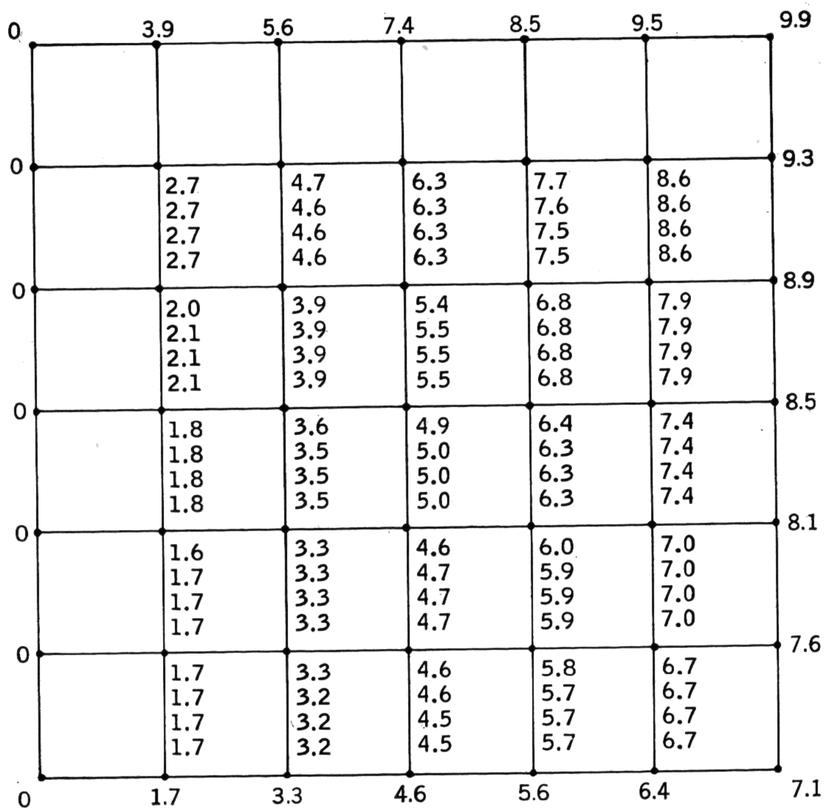


FIG. 9-23. A finer lattice calculation than that of Fig. 9-22. The boundary values were taken from Fig. 9-21.

For our example, the difference equation (3) can be written as

$$(u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}) - 4u_{ij} = 0$$

[where we have replaced $u(x + h, h)$ by $u_{i,j+1}$, $u(x - h, y)$ by $u_{i,j-1}$, $u(x, y + h)$ by $u_{i+1,j}$, and $u(x, y - h)$ by $u_{i-1,j}$]. Thus the residual at point (i,j) can be expressed as

$$R_{ij} = (u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}) - 4u_{ij} \tag{4}$$

and the purpose of the relaxation method is to try to make the residual at every point become equal to zero. This is accomplished by appropriately adjusting the value of u_{ij}. Two problems therefore remain for discussion: first, by how much should u_{ij} be adjusted, and second, after u_{ij} is adjusted, how are the residuals at neighboring points corrected to account for this adjustment?

If in our example we adjust the value of u_{ij} by Δu_{ij}, then from (4) the residual R_{ij} is altered by an amount ΔR_{ij} given by

$$\Delta R_{ij} = -4\,\Delta u_{ij}$$

But we desire that the final residual be zero, i.e., that $R_{ij} + \Delta R_{ij} = 0$, whence

$$R_{ij} = -\Delta R_{ij}$$

and we must therefore adjust u_{ij} by an amount

$$\Delta u_{ij} = \tfrac{1}{4} R_{ij}$$

which solves the first of the problems mentioned above. Next consider the new value of the residual of an adjacent point, say $R_{i,j+1}$. This residual, which was originally

$$R_{i,j+1} = (u_{i,j+2} + u_{i,j} + u_{i+1,j+1} + u_{i-1,j+1}) - u_{i,j+1}$$

becomes now

$$R_{i,j+1} + \Delta R_{i,j+1} = [u_{i,j+2} + (u_{i,j} + \Delta u_{ij}) + u_{i+1} + u_{i-1,j+1}] - u_{i,j+1}$$

Hence

$$\Delta R_{i,j+1} = \Delta u_{ij}$$

giving the needed change in residual of the adjacent point. Similarly for the other adjacent points, $\Delta R_{i,j-1}$, $\Delta R_{i+1,j}$, and $\Delta R_{i-1,j}$ are all equal to Δu_{ij}.

Now we can completely describe the method of relaxation as follows: First, compute the residual at each lattice point. Second, choose the point with the largest residual, and adjust the value of that point according to a formula derived from the associated difference equations so that the residual there becomes zero. Third, recalculate the *residuals* at the adjacent points to account for the new value of the adjusted point. Then the residuals are again scanned and the largest residual again chosen, and the process continued until all the residuals become zero at the same time. This process of *relaxation* is accomplished for each fineness of array lattice.

9-11. Accuracy and Error

Sources of Error. We can distinguish among three contributions to the error of a numerical computation:

1. First is the error due to the approximate nature of the numerical analysis of the computations. As we have seen in previous sections, frequently the value of a function or the solution to an equation can theoretically be obtained only by performing an infinite sequence of iterations. Of course the infinite process cannot be completed, and so we must stop with some iteration in the sequence, accepting this as an adequate approximation to the desired results. The resulting error

is called a *truncated error* and often can be made as small as we please. *The problem here then is to estimate the accuracy of the result after an iteration to determine whether or not it is as precise as required, i.e., whether or not more iterations should be made.*

2. Second is the error due to the approximate nature of experimental data used in the calculation; this is called the *experimental error. Here the problem is so to analyze the experimental data as to obtain the best possible estimate of the measurement involved,* i.e., to determine the accuracy of the experimentally obtained numbers. The accuracy of such numbers depends on aspects other than mathematical considerations, and therefore from a computational point of view these errors are treated as initial constraints on the problem.

3. Third is the error due to the use of approximate numbers in performing the numerical calculations; such errors are called *round-off errors.* Any number that we write is necessarily in finite decimal (or binary) form, which is only an approximation to its infinite decimal conception.† For example, π might be approximated by 3.1415927. Numbers can be approximated to any degree of accuracy: e.g., a better approximation to π is 3.14159 26535 89793. Experimental numbers may also be considered to be finite approximations of some (unknown) infinite decimal. Here of course we cannot get any better approximation than those determined by methods associated with the above-discussed second source of error. *Thus the problem here is to perform the four arithmetic operations in a manner that retains the maximum desired accuracy under the circumstances.*

Consideration of truncated errors forms a large subject in itself. For the various iterative formulas given in the previous sections, formulas can be obtained for the errors, or for bounds on the errors, after any particular iteration. For example, the remainder formula for a Taylor series is familiar to the student as a measure of the error of approximation of a finite number of terms of a power series. Textbooks on numerical analysis consider these formulas in detail; this subject is, however, beyond the scope of the present brief treatment. Consideration of experimental errors is also beyond the scope of this treatment.

This section then considers the last of these three problems, the round-off errors. The importance of this subject to numerical calculators cannot be overestimated. During a computation it is not an overstatement to say that consideration of errors that may arise from arithmetic operations should be uppermost in the mind of the user of the computer. The discussions of this section merely form the basis for such error considerations in a particular computation. Very often it is necessary for the user of the computer to determine experimentally the behavior of his numbers during the computations, by having the computer perform

† Here we consider, for example, the number 0.5000 · · · to be an infinite decimal.

series of trial computations. These experimental computations can be used to determine the correct operational procedures.

Significant Figures and Round Numbers. The accuracy of a number is measured in terms of the number of significant figures it contains. Any one of the digits 1, 2, 3, 4, 5, 6, 7, 8, 9 is a significant figure; 0 is sometimes but not always a significant figure. To determine when 0 is a significant figure, we distinguish three cases:

1. The zero lies to the left of all other nonzero digits of the number, e.g., as in 0.00123. Here the zero is used simply to fix the decimal point, and therefore it is *not* considered to be significant. None of the three zeros in 0.00123 is significant.

2. The zero or zeros lie between nonzero digits, e.g., as in 3,002. Here the zeros are considered to be significant.

3. The zero lies to the right of all other nonzero digits, e.g., 123,000. In this case it cannot be determined from the number itself whether or not the zero is significant; it must be determined from the context in which it was written. Often, if the zeros in 123,000 are not significant, we would write 123×10^3, while on the other hand if the first zero were significant we would write $1{,}230 \times 10^2$ to indicate this.

For example, the *number of significant figures* in each of the following numbers is four: 1,234, 9,072, 5,006, 0.008379, 0.01024, $6{,}080 \times 10^3$, $7{,}000 \times 10^3$. The leftmost significant figure in a number is called its *most significant figure;* in our examples the most significant figures are, respectively, 1, 9, 5, 8, 1, 6, and 7. The *least significant figure* is the rightmost significant figure of a number; in our examples the least significant figures are, respectively, 4, 2, 6, 9, 4, 0, and 0.

A number composed of n significant figures is said to be *correct to n significant figures* if its value is correct to within $\frac{1}{2}$ unit in the least significant position. For example, if 9,072 is correct to four significant figures, then it is understood that the number lies between 9,072.5 and 9,071.5 (that is, $9{,}072 \pm 0.5$); if 0.01024 is correct to four significant figures, then it lies between 0.010245 and 0.010235; if $6{,}080 \times 10^3$ is correct to four significant figures, it lies between 6,080,500 and 6,079,500.

Another term used to describe a number is the *number of decimal places* of the number, being the number of digits to the right of the decimal point. Thus 0.008379 has six decimal places, and 0.37×10^{-5} has seven decimal places.

To be used during a computation, numbers that are infinite decimals must be converted to finite-decimal approximations. Similarly numbers that contain more significant figures than are desirable for a particular computation must be approximated by numbers of fewer significant figures. The process of reducing the number of significant figures of a number is called *rounding.* Numbers that are rounded so that they contain n

significant figures are formed so that they will be *correct to n significant figures;* they are then called *round numbers.* Rounding can be accomplished by means of the following rule: To round a number so that it is correct to n significant figures, retain the n leftmost significant figures, and discard the rest; if the discarded number is greater than ½ unit in the least retained significant figure, then add 1 to this last retained figure; if the discarded number is less than ½ unit in the least retained significant figure, leave the latter unchanged. The case where the discarded number is exactly ½ unit in the least retained significant figure is often handled thus: If the least significant figure is even, leave it unchanged; if odd, then add 1 to it. By means of this latter rule the errors due to rounding will tend to cancel, since even and odd digits occur on the average with equal frequency. Also, by leaving the rounded number in this case as an even number we increase our chances that the result of a division might be exact. As examples, the following numbers have been rounded to four significant figures:

Number	*Rounded number*
3.141|5926	3.142
0.1428|57142	0.1429
0.005329|601	0.005330
421.2|4899	421.2
58.76|5000	58.76
93.29|5000	93.30

Relative Accuracy and Absolute Accuracy. The error in a number, as compared with its true value, is stated in various ways. The *absolute error* of an approximate number N^* from its true value N is defined by

$$\epsilon_a = |N^* - N|$$

The relative error ϵ_r is defined by

$$\epsilon_r = \frac{|N^* - N|}{N^*} = \frac{\epsilon_a}{N^*}$$

The percentage error ϵ_p is defined by $\epsilon_r \times 100$ per cent. The relative- and percentage-error concepts have the advantage that they are independent of the unit of measurement, whereas the absolute error is expressed in terms of the unit used. For example, 3.14 is an approximate value of the infinite decimal π; the absolute error of this approximation is $|3.14 - \pi| < 0.002$ $(\pi - 3.14 = 0.00159 \cdots)$; the relative error is $|3.14 - \pi|/3.14 < 0.002/3.14 < 0.0006$; the percentage error $\epsilon_p < 0.06$ per cent.

The accuracy of an approximate number can be considered from two points of view. The first, and that most often considered in using digital computers, is the concept of *relative accuracy.* In this case the accuracy

is measured by the *relative* error and not the absolute error. As will be shown two paragraphs below, the number of *significant figures* of an approximate number is a measure of the relative error and hence of the relative accuracy of the number. It is for this reason that significant figures are so important. To see the importance of relative error as an index of the accuracy of a number, consider the measurement of the 0.002-in. diameter of a human hair to within 1/100,000 in., as compared with the measurement of the 240,000-mile distance from the earth to the moon, to within 2 miles. The relative error of the measurement of the hair is 0.00001/0.002 = 1/200, and the relative error of the measurement of the distance to the moon is 2/240,000 = 1/120,000, while the absolute errors are 0.00001 and 126,720 in., respectively. Clearly the measurement of the distance to the moon is 600 times more accurate than that of the diameter of the human hair, even though the absolute error of the former is over 12×10^9 times that of the latter.

On the other hand, the *absolute accuracy* of a measurement is often important from another point of view. For example, in the tolerance of machined parts it is the absolute error that counts as the measure of accuracy; hence the allowed absolute error is frequently stated explicitly by writing the desired sizes as finite decimals followed by ± and the tolerance which is the maximum allowable absolute error. In this case it is the number of decimal places that plays the important role.

The relation between the number of significant figures and the relative error inherent in an approximate number is given by the following theorem, in which ϵ_r is the relative error, K is the most significant figure of the number, and n is a positive integer:

If a given number is correct to n significant figures, then

$$\epsilon_r \leq \frac{1}{2K \times 10^{n-1}}$$

When the most significant figure is not explicitly known, we can say that the relative error $\epsilon_r \leq 1/(2 \times 10^{n-1})$. For any radix q the 10^{n-1} in these formulas is replaced by q^{n-1}. If $q = 2$ (binary), then the inequalities are read $\epsilon_r \leq 1/2^n$.

To see why this theorem is true, note that $N^* \geq K \times 10^{p-1}$ (for example, $521 \geq 500 = 5 \times 10^2$), where p locates the decimal point with respect to the most significant figure K.† Now since N^* represents an approximate number, correct to n significant figures, i.e., with an error of ½ unit in the last (nth) significant figure, then $\epsilon_a \leq \frac{1}{2} \times 10^{p-n}$ and

$$\epsilon_r \leq \frac{\frac{1}{2} \times 10^{p-n}}{N^*} = \frac{1}{2N^* \times 10^{n-p}} \leq \frac{1}{2K \times 10^{n-1}}$$

† For example, 3,000 would be written $3 \times 10^3 = 3 \times 10^{4-1}$, whence $p = 4$. On the other hand, 0.003 would be written $3 \times 10^{-3} = 3 \times 10^{-2-1}$, whence $p = -2$.

If K is not known, then to ensure that the inequality will hold for every K we choose the smallest, namely, 1.

This theorem is of important practical value in using a digital computer having floating-point arithmetic operations. It enables the evaluation of the number of significant figures that must be carried to ensure that the relative error be kept below a given amount; and conversely it enables the evaluation of the relative error for a specified number of significant figures.

Significant Figures in Arithmetic Operations. Since numerical computation as performed by a digital computer reduces to addition, subtraction, multiplication, and division, the problem of retaining significance during these operations becomes fundamental. We shall here give rules of thumb commonly used by mathematicians regarding the control of significance during arithmetic operations. In any particular situation the rules serve merely as a guide; the details of the process usually depend on the particular numbers and errors involved and hence must often be carefully examined. *In many situations it is just as bad to overestimate an error resulting from a computation as it is to underestimate one.*

Consider addition first. The rule of thumb is that in adding two (positive) numbers there is usually no loss of significant figures; in fact the result sometimes has one more significant figure than either of the operands.

For example, suppose that 52,761.2 and 71,436.7 were rounded to 52,761 and 71,437 and added:

Rounded	*Unrounded*
52,761	52,761.2
+ 71,437	+ 71,436.7
124,198	124,197.9

The rounded numbers are both correct to five significant figures, and the result is correct to six, as is seen on comparison with the addition of the unrounded numbers. Here there was a gain of significance, because the rounding errors canceled out, and the added significant figure resulted from the carry. However, consider 12,761.5 and 11,435.5 rounded to 12,762 and 11,436 and added:

Rounded	*Unrounded*
12,762	12,761.5
+11,436	+11,435.5
24,198	24,197.0

Here the result is correct only to four significant figures since, as can be seen from the unrounded addition, the rightmost 8 is off by 1 unit. This

is of course an exception to our rule, chosen so that the errors in the least significant figures of the operands were maximum and were added together. It is clear that, if one always includes one more significant figure than is necessary, one will be more sure of having at least the required significance in one's result.

Subtraction, however, is another story. The rule is simply a warning that complete loss of significance is possible and that each case must be examined on its own merits. For example, suppose that we are forming the difference of 52,763.5 and 52,762.5 as rounded to 52,764 and 52,762:

Rounded	*Unrounded*
52,764	52,763.5
−52,762	−52,762.5
00,002	00,001.0

Although the subtrahend and the minuend were correct to five significant figures, the difference has no significant figures; i.e., the 2 is off by a full unit. Most generally there is not a complete loss of significance, but very frequently severe loss of significance occurs. There are two remedies that can be tried. The first is to use many more significant figures in the subtrahend and minuend to begin with and hope that enough significance will result for the remainder. The other is to try to transform the expression to something else and thereby avoid the subtraction altogether. For example, if one had to evaluate $1 - \cos x$ for small x, it might be wiser to write $1 - \cos x = 2 \sin^2 (x/2)$ and evaluate the latter expression.

For multiplication the rule is that up to two significant figures can be lost, though not necessarily. This same rule applies to division. Consider, for example, the product of 811.2 and 112.5, each rounded to three significant figures:

Rounded	*Unrounded*
811	811.2
112	112.5
1622	40560
811	16224
811	8112
90832	8112
	91260.00

The third figure of the product is off by 4 units and therefore is not significant; the second figure from the left is, however, off by only 0.4 unit and therefore is significant. The product is thus correct to two significant figures, while the multiplicand and multiplier were each correct to three.

As an example of the loss of a significant figure during division, consider 1,761.5 and 1,763.4, rounded to 1,762 and 1,763:

Rounded	*Unrounded*
1.0005	1.001
1,762|1,763.0000	17,615|17,634.000
1,762	17,615.000
1.0000	19.000

Here the rounded dividend and divisor are correct to four significant figures, but the quotient is correct only to three.

One further rule might be helpful. In digital computers, where it is in general no more difficult to handle numbers with many significant figures, keep as many significant figures as possible during the computations, and beware of loss of significance through subtraction.

Pitfalls. A person using an electronic computer always worries about whether the results he gets are or are not accurate. For usually only the final results of a computation are read out from the computer's memory, and intermediate results are neither retained nor observed. The results of computer programs are often checked by comparing a few values with some previously hand-computed values. However, the fact that these few isolated results check is no guarantee that all the results computed by the program will be equally correct. Rarely is every difficulty fully anticipated. Automatic programming routines aggravate this situation, increasing the possibility of overlooking pitfalls. Hence, when writing his program the programmer must plan against the loss of too many significant figures in the computation, as well as choose the computational method that can best result in the necessary accuracy. During the running of the program on the computer the programmer must be acutely aware of such pitfalls as overflows, loss of significance, difficulties arising in intermediate forms, and other, unanticipated, singular situations.

For example, consider a pitfall that occurs in evaluating $\sqrt{x}$. If $x = 0.0000\ 3259$, where x is good to seven decimal places and uncertain in the last (i.e., the 9), then to eight decimal places we get

$$\begin{aligned}\sqrt{0.0000\ 3258} &= 0.0057\ 0789\\ \sqrt{0.0000\ 3259} &= 0.0057\ 0877\\ \sqrt{0.0000\ 3260} &= 0.0057\ 0964\end{aligned}$$

Hence $\sqrt{x}$ is good only to five places, is uncertain in the sixth place, and is erroneous in the last two places. However, the computer will carry these last two places, and the error may be magnified in subsequent calculations.

Another example occurs in computing, say, sin 314,159.3. For if we reduce the angle to one whose magnitude is less than $\pi/2$, we find

$$\sin 314{,}159.3 = \sin (314{,}159.3 - 314{,}159.27) = \sin 0.03 = 0.03$$

If the last figure of the original angle were 2 instead of 3, we would find sin 314,159.2 = −0.07. If 4, we would find sin 314,159.4 = +0.13. Hence, even though the original angle is known to within seven significant figures with an uncertainty of 1 in the last figure, the result is not good to within even one decimal place.

The following quotation† summarizes the general attitude that should be maintained in performing numerical computations on a digital computer:

> Despite the availability of high speed computers, a successful computation depends on a judicious combination of mathematical analysis and numerical experimentation. In the actual programming of his problem, the numerical analyst should not use subroutines and automatic codes blindly, but should ascertain that the logical decisions made in them will be consistent with the required round-off error and tolerance limits. The possibility of pitfalls will always be present in spite of all efforts to avoid them. It is therefore important that the numerical analyst interpret his answers according to their true meaning. Although the tools of the numerical analysis are all based on rigorous mathematical theorems the application of these theorems requires discretion. In carrying out a numerical computation program it is often necessary to use experimental techniques to establish the correct operational procedures.

9-12. Exercises

9-1. Solve

$$\begin{aligned}5x_1 - 2x_2 &= 3\\ 4x_1 - 5x_2 &= -18\end{aligned}$$

by following in detail the steps of the flow diagram, Fig. 9-1, keeping track of all the tallies.

9-2. Solve

$$\begin{aligned}0 \cdot x_1 + 4x_2 + x_3 - 3x_4 &= 1\\ 4x_1 + 0 \cdot x_2 - 2x_3 + 2x_4 &= 14\\ 8x_1 - 7x_2 + x_3 - 3x_4 &= -12\\ -5x_1 + 8x_2 + 0 \cdot x_3 - x_4 &= -2\end{aligned}$$

Solution: $x_1 = 8,\ x_2 = 7,\ x_3 = 27,\ x_4 = 18$.

9-3. The flow chart of Fig. 9-1 does not account for zero coefficient in the leading term of Eq. $(j)^i$. How can the flow chart be adjusted to account for this?

9-4. Using the three-address instructions of Chap. 3, write a code that solves simultaneous linear equations.

† Irene A. Stegun and Milton Abramowitz, Pitfalls in Computation, *J. Soc. Ind. Appl. Math.*, vol. 4, no. 4, December, 1956.

Solve by successive approximations, to within 0.001 (*decimal*):

9-5. $1.5 \cos x + 2x - 2 = 0$.

9-6. $\sin x - \tan^2 x = 0$.

9-7. $\dfrac{1 - x^2}{x} - \dfrac{x}{1 - x^2} = 0$. (HINT: There are four solutions.)

9-8. $\dfrac{1 - x^2}{x} - \tan^2 x = 0$.

Consider the equation $x = \frac{1}{2}(e^{\alpha x} - e^{-\alpha x}) = \sinh \alpha x$. Take $x_0 = 1$, and show by direct computation:

9-9. If $\alpha = -1.2$, the ordinary iterative process will be of type 2 but the accelerating process will converge.

9-10. If $\alpha = 0.5$, the ordinary iterative process will be of type 3 but the accelerating process will make the convergence more rapid.

9-11. If $\alpha = 1.2$, the ordinary iterative process will be of type 4 but the accelerating process will converge.

9-12. In Fig. 9-6, on the right-hand sides of the equations for m and x^*_{n+1}, we must use x_{n+1} since x^*_{n+1} is not yet calculated; however, x^*_n and x^*_{n-1} can be used. Why, then, does x_n rather than x^*_n appear in the numerator of the expression for m?

9-13. In the second series expansion given in Sec. 9-3 for arctan x, how many terms are needed to determine arctan 0.5 to within ± 0.001?

9-14. Using the continued-fraction expansion of arctan x, compute the value of π to seven significant figures. (HINT: arctan $1 = \pi/4$.)

9-15. Draw a flow chart for evaluating the nth convergent of a continued fraction.

9-16. Construct a difference table for the polynomial

$$y = 2x^4 + 3x^2 + 5x + 2$$

with $h = 1$.

9-17. Introduce an error $\epsilon = 2$ in one of the values y in the above polynomial; recompute the difference table, and observe how the error propagates.

9-18. Show that $f^*_{0,1,2,3}(x)$ actually does pass through all the points (x_0,y_0), (x_1,y_1), (x_2,y_2), (x_3,y_3) and is of degree 3.

9-19. Show that

$$f^*_{0,1,2}(x) = \frac{\begin{vmatrix} f^*_{0,1}(x) & x_0 - x \\ f^*_{1,2}(x) & x_2 - x \end{vmatrix}}{x_2 - x_0} = \frac{\begin{vmatrix} f^*_{0,2}(x) & x_0 - x \\ f^*_{1,2}(x) & x_1 - x \end{vmatrix}}{x_1 - x_0}$$

9-20. Compute arctan 0.9, where arctan x is given by a table for values of x differing by 0.4, as shown:

x_i	arctan x_i	Solution					
$x_0 = -1.0$	−0.7854	−1.9					
$x_1 = -0.6$	−0.5404	−1.5	0.3783				
$x_2 = -0.2$	−0.1974	−1.1	0.6111	1.2515			
$x_3 = +0.2$	+0.1974	−0.7	0.7707	1.1141	0.8736		
$x_4 = +0.6$	+0.5404	−0.3	0.7890	0.8917	0.7568	0.6691	
$x_5 = +1.0$	+0.7854	+0.1	0.7068	0.6863	0.7334	0.7509	0.7305

9-21. Compute arctan 0.75 from the table values given in the previous exercise.

9-22, 9-23. Write a code to evaluate each of the best-fit approximations to arctan x given in Sec. 9-5.

The following table gives the values of arctan x for equally spaced points:

x_i	*arctan* x_i
$x_0 = 0$	0.00000
$x_1 = 0.2$	0.19739
$x_2 = 0.4$	0.38051
$x_3 = 0.6$	0.54042
$x_4 = 0.8$	0.67474
$x_5 = 1.0$	0.78540

9-24. Evaluate $\int_0^{0.8} \arctan x\, dx$ by the rectangular rule, first for partition at x_0, x_2, and x_4, and then for partition at x_0, x_1, x_2, x_3, and x_4; compare the results.

9-25. Evaluate $\int_0^{0.8} \arctan x\, dx$ by the trapezoidal rule, first for partition at x_0, x_2, and x_4, and then for partition at x_0, x_1, x_2, x_3, and x_4; compare the results.

9-26. Evaluate $\int_0^{0.8} \arctan x\, dx$ by Simpson's rule, first for partition at x_0, x_2, and x_4, and then for partition at x_0, x_1, x_2, x_3, and x_4; compare the results.

9-27. Integrate each of the three polynomial approximations for arctan x given in Sec. 9-5, evaluate each of these integrals from 0 to 0.8, and compare the results of each with the results of Exercises 9-24 to 9-26.

9-28. Observing that (the indefinite integral)

$$\int \arctan x\, dx = x \arctan x - \tfrac{1}{2} \ln (1 + x^2)$$

find $\int_0^{0.8} \arctan x\, dx$; compare this result with those of Exercises 9-24 to 9-27.

9-29. Write a code for evaluating Simpson's rule, assuming that $f(x)$ is evaluated by a subroutine.

9-30. Determine the derivative of arctan x at $x = 0.6$, using first Δx between the two points at $x = 0.4$ and $x = 0.6$, and then between the points at $x = 0.6$ and $x = 0.8$.

9-31. Determine the derivative of arctan x at $x = 0.6$ by using the three points first at $x_0 = 0.6$, $x_1 = 0.8$, and $x_2 = 1.0$, then at $x_0 = 0.4$, $x_1 = 0.6$, and $x_2 = 0.8$, and finally at $x_0 = 0.2$, $x_1 = 0.4$, and $x_2 = 0.6$.

9-32. Differentiate each of the three polynomial approximations for arctan x given in Sec. 9-5, and in each case evaluate the derivative at $x = 0.6$.

9-33. Using the form

$$\frac{d}{dx} \arctan x = \frac{1}{1 + x^2}$$

evaluate the derivative of arctan x at $x = 0.6$, and compare this with the approximations of Exercises 9-30 to 9-32.

9-34. Show that, if (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3) are four points with equally spaced abscissas, then

$$\int_{x_1}^{x_2} y\, dx = \frac{h}{24}(-y_0 + 13y_1 + 13y_2 - y_3)$$

(HINT: Let $x_0 = -\frac{3}{2}h$, $x_1 = -h/2$, $x_2 = h/2$, and $x_3 = \frac{3}{2}h$; let the four functions be $y = 1$, $y = x$, $y = x^2$, and $y = x^3$.)

9-35. If five points with equally spaced abscissas are given, (x_0,y_0), (x_1,y_1), (x_2,y_2), (x_3,y_3), and (x_4,y_4), then find $\int_{x_1}^{x_3} y\,dx$.

9-36. Derive the formulas given in Sec. 9-7 by the method of undetermined coefficients.

9-37. If four points with equally spaced abscissas, (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3), are given, find formulas for

$$\left(\frac{dy}{dx}\right)_{x=x_0} \qquad \left(\frac{dy}{dx}\right)_{x=x_1} \qquad \left(\frac{dy}{dx}\right)_{x=x_2} \qquad \left(\frac{dy}{dx}\right)_{x=x_3}$$

9-38. Find the formula for $\int_{x_0}^{x_1} y\,dx$ in terms of the two points (x_0,y_0) and (x_1,y_1), and of the derivatives at these points,

$$y_0' = \left(\frac{dy}{dx}\right)_{x=x_0} \qquad \text{and} \qquad y_1' = \left(\frac{dy}{dx}\right)_{x=x_1}$$

(HINT: Let $x_0 = 0$, $x_1 = h$; let

$$\int_{x_0}^{x_1} y\,dx = A_0y_0 + A_1y_1 + B_0y_0' + B_1y_1'$$

where A_0, A_1, B_0, and B_1 are undetermined coefficients. Now, when the functions $y = 1$, $y = x$, $y = x^2$, and $y = x^3$ are substituted successively into the equation, there result four simultaneous equations that determine A_0, A_1, B_0, and B_1.)

9-39. Solve $dy/dx = 1 + y^2$ for $h = 0.01$ and $h = 0.001$ by the methods of Sec. 9-9, and compare with a table of tan x.

9-40. Solve $dy/dx = \sqrt{1 - y^2}$ by the methods of Sec. 9-9 with intervals $h = 0.1$ and 0.01, where $x_0 = 0$, $y_0 = 0$. Compare the solutions with a table of sin x.

9-41. Draw a flow chart for Runge's method for solving differential equations.

9-42. Write the difference equations corresponding to

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x,y) \qquad \text{and} \qquad \frac{\partial u}{\partial x} = c^2\frac{\partial^2 u}{\partial y^2}$$

9-43. Using the boundary values given in Fig. 9-23, solve Laplace's equation by the relaxation method for 4 and then for 25 interior lattice points.

9-44. Tell how many significant figures are in each of the following numbers, and round each to be correct to five significant figures (or bits):

2.7182818	1.11001110 (*binary*)
7.3890568	.00110010 $\times 2^{-3}$ (*binary*)
0.3183099	101011.00 (*binary*)
0.3678794	0011001100 $\times 2^3$ (*binary*)
484,813,681,110 $\times 10^{-6}$	1010101010 (*binary*)
154.1500 $\times 10^3$	11.001100 (*binary*)

9-45. Calculate the maximum relative error of each of the above numbers, rounded to three significant figures (or bits in the case of binary numbers).

9-46. If all the numbers in a computation were kept to five significant figures, what is a bound on the relative error of each number? If these numbers were binary numbers, what is the bound on the relative error?

9-47. If it is desired that all numbers involved in a computation have a relative error of less than $\frac{1}{600}$, how many significant figures should each number contain?

9-48. In a computation involving only binary numbers, if it is desired that all numbers should have a relative error of less than $\frac{1}{600}$, how many significant bits should each number contain?

9-49. *General Formula for Measurement Errors.* Suppose that $N = f(x_1,x_2, \ldots ,x_n)$ denotes any function of n independent variables x_1, x_2, . . . , x_n that are subject to the experimental errors Δx_1, Δx_2, . . . , Δx_n. Show that a formula giving the error in the function N is

$$\Delta N = \frac{\partial N}{\partial x_1}\Delta x_1 + \frac{\partial N}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial N}{\partial x_3}\Delta x_3$$

Suppose that the error of the independent variables is not actually known, but it is known that the probability distribution of the errors is *normal* (Gaussian). Then we can define the *probable error* of a single measurement as a value such that half the errors of a series of measurements will be greater, the other half less. If R, r_1, r_2, . . . , r_n are the probable errors, respectively, of N, x_1, x_2, . . . , x_n, then show that

$$R = \sqrt{\left(\frac{\partial N}{\partial x_1}\right)^2 r_1^2 + \left(\frac{\partial N}{\partial x_2}\right)^2 r_2^2 + \cdots + \left(\frac{\partial N}{\partial x_n}\right)^2 r_n^2}$$

For further discussion see Scarborough (see the References below), chaps. XIV–XV, and E. Frank, "Electrical Measurement Analysis," chaps. 5–7, McGraw-Hill Book Company, Inc., New York, 1959.

9-50. *Error Formulas for Differentiation and Integration.* Milne ("Numerical Calculus," pp. 108–114; see the References below) gives a method for deriving the error formula for any differentiation-integration method that may be developed by the method of undetermined coefficients. If $Q(f)$ is the integration or differentiation to be performed on the function f and $P(f)$ is the formula developed by the method of undetermined coefficients for its approximation, then the error $R(f) = Q(f) - P(f)$. $R(f)$ is said to be of *degree* n when $R(x^m) = 0$ for $m \leq n$, but $R(x^{n+1}) \neq 0$. Also let $\overline{(x - s)}^n$ be defined by $\overline{(x - s)}^n = (x - s)^n$ if $x > s$ and $\overline{(x - s)}^n = 0$ if $x < s$. In terms of this definition let

$$G(s) = \frac{1}{n!}R_x[\overline{(x - s)}^n]$$

in which $R_x[\overline{(x - s)}^n]$ means $R[\overline{(x - s)}^n]$ regarded as a function of the variable x. Then it can be shown that

$$R(f) = \int_{-\infty}^{\infty} f^{n+1}(s)G(s)\,ds$$

where f^{n+1} is the $(n + 1)$st derivative of f (see Milne for an example). Using this technique, derive error formulas for the integration and differentiation formulas of Secs. 9-6 to 9-8.

REFERENCES

Alt, Franz L.: "Electronic Digital Computers," Academic Press, Inc., New York 1958.

Engineering Research Associates, Inc., Staff of: "High-speed Computing Devices," McGraw-Hill Book Company, Inc., New York, 1950.

Grabbe, E. M., S. Ramo, and D. E. Wooldridge: "Handbook of Automation, Computation, and Control," vol. 1, John Wiley & Sons, Inc., New York, 1958.

Hastings, Cecil, Jr.: "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955.
Hildebrand, F. B.: "Introduction to Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1956.
Householder, A. S.: "Principles of Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1953.
Lanczos, C.: "Applied Analysis," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1956.
Levy, C., and B. A. Baggott: "Numerical Solutions of Differential Equations," Dover Publications, Inc., New York, 1950.
Milne, W. E.: "Numerical Calculus," Princeton University Press, Princeton, N.J., 1949.
———: "Numerical Solution of Differential Equations," John Wiley & Sons, Inc., New York, 1953.
Nielsen, Kaj L.: "Methods in Numerical Analysis," The Macmillan Company, New York, 1956.
Scarborough, J. B.: "Numerical Mathematical Analysis," 2d ed., Johns Hopkins Press, Baltimore, 1950.

CHAPTER 10

BOOLEAN ALGEBRA

10-1. Introduction

There is perhaps too little appreciation of the great number and variety of problems that can be attacked and solved by methods of the propositional calculus of symbolic logic and the related Boolean algebra, which is the calculus of sets and classes. Direct application of logic can always be an aid to deductive reasoning—such as determining consequences of given premises, rules, or axioms and making hypotheses or theorems from which the given premises or factual relationships can be deduced. Besides the use of logical propositional methods in problems concerned with sentences, such as analysis of military information reports and legal and insurance documents, there appear to be even more important applications to fields of operations research, biology, medicine, design of experiments, etc., where the utility of symbolic logic per se is not so immediately evident. In this chapter we present digital methods which can aid in making many types of logical problems computationally feasible. It is hoped that these methods can be directed toward the successful solution of many such problems—two illustrative examples being given at the end of this chapter. The digitalized methods which use the binary number system provide easy mechanization on existing high-speed electronic computers.

Historical Background. It is of historical interest to note that this algebra was first investigated by George Boole (1815–1864) and the theory essentially completed by Ernst Schroeder in his monumental work on the algebra of logic before the turn of this century. Boole, Schroeder, and other logicians of that time had in mind the practical uses of the algebra of logic in the solution of everyday problems. However, they found that dealing with just four to five variables made practical computation unfeasible, requiring the employment of extremely ingenious methods for solution. Hence interest in solving large-scale logical problems of this kind was lost.

It was at that time, too, that Whitehead and Russell, Hilbert, and others realized that logic could be developed to encompass the foundations of mathematics, and after the turn of the century the attention of

logicians became focused on the so-called *functional calculus of symbolic logic.* Problems of extreme metamathematical and philosophical importance were considered. Such problems are still today of primary interest to professional logicians. Their methods have necessarily been esoteric and have as yet found little everyday application.

On the other hand, the advent of digital-computer circuitry brought a widespread reawakening of interest in the algebra discovered by Boole. Computational difficulties again limited the direct, practical application of Boolean algebra. To help offset some of these difficulties, digital computational methods have recently been developed, and some of these are presented in this chapter.

10-2. Definition of Logic and Propositions

Symbolic Logic. Logic is the science that teaches how thinking or reasoning should proceed; it produces the results of thinking; and it encompasses laws by means of which a judgment can be passed as to the correctness of the products of thought or reasoning. As H. Reichenbach points out, however, "When we call logic *analysis of thought,* the expression should be interpreted so as to leave no doubt that it is not actual thought which we pretend to analyze, it is rather a substitute for thinking processes, their *rational reconstruction,* which constitutes the basis of logical analysis." In other words, once a result of thinking or reasoning is obtained, a chain of thought can be constructed from the initial premises to the final conclusions and an analysis of this rational reconstruction of thinking reveals those rules called laws of logic. Logic presents a means by which thinking or reasoning may be simulated and the results of thinking obtained or checked.

Only when the results of reasoning have been stated in terms of sentences can they be tested or evaluated; logic deals with sentences (e.g., statements) and the rules correlating premise and conclusion sentences. The best method of handling these sentences in logic is a symbolic method, for while simple logical operations can be performed without the help of a symbolic notation, the structure of complicated relations cannot be seen without the use of symbols.

The Proposition. The methods of symbolic logic that are concerned with deduction must lead from true sentences to true sentences. Hence the whole sentence becomes a fundamental unit for logic, since only a whole sentence can be *true or false.* In fact, attention in this book will be limited only to whole sentences with this property, and such sentences will be called propositions. *A proposition is a whole sentence about which it is meaningful to say that its content is true or false.* (Some authors put this somewhat differently, limiting their consideration only to meaningful sentences. In their system a sentence is meaningful only if it is *verifiable*

as true or false.) The part of symbolic logic that deals with propositions is called the *propositional calculus.*

Examples of propositions and their truth values are as follows:

	Proposition	*Truth value*
a.	3 is an even number	False
b.	The grass is green	True
c.	Logic is the science that teaches how thinking or reasoning should proceed	True
d.	The proposition on the line below is true	False
e.	January is a summer month	False

In textbooks dedicated solely to symbolic logic a distinction in symbolism is made between propositions and propositions about propositions. However, since the purpose of this chapter is only to introduce computational methods in the propositional calculus of symbolic logic and not to expound the subject itself, no consistent attempt will be made to separate propositions of these two types. The distinction between propositions of these types is to be made by the reader from the context.

The Propositional Operations. It is to be reemphasized that this chapter deals with a logic of two truth values, viz., *true* and *false,* and that the primary interest is in determining the truth value of propositions. In considering combinations of propositions, the truth value of the combination is again the primary object.

Propositions can be combined or manipulated by the words *and, or,* and *not* (negation), these words being called the *propositional operations.* The operations *and* and *or* combine two propositions, whereas the operation *not* (or negation) is associated with only one proposition. Given the truth value of propositions, the truth values of their combinations with the propositional operations are defined as follows:

1. The *and* combination of two propositions is true if and only if both propositions are true. If either one or both of them are false, the combination is false. For example, "The grass is green, *and* logic is the science that teaches how thinking or reasoning should proceed" is true, but "The grass is green, *and* 3 is an even number" is false. The *and* combination of two propositions is often called their *conjunction,* or *logical product.*

2. The *or* combination of two propositions is true if and only if either one or both of the propositions are true. The *or* combination of two propositions is false only if both propositions are false. For example, "The grass is green, *or* 3 is an even number" is true; "3 is an even number, *or* January is a summer month" is false. The *or* combination of two propositions is often called their *disjunction,* or *logical sum.*

3. The negation of a proposition is true if and only if the proposition is false. For example, "3 is *not* an even number" (which is the negation of "3 is an even number") is true.

Propositions may be symbolized by letters: A, B, C, . . . , W, X, Y, ... In later sections we shall symbolize propositions by letters with subscripts: A_1, A_2, . . . , B_1, B_2, ... We use the symbol $\cdot$ for the operation *and*, $+$ for *or*, and $^-$ (bar) over the letter to indicate negation. This symbolism is summarized in the following table:

Operational symbol	Interpretation	Terminology
$\bar{A}$	*Not A*	Negation
$A + B$	*A or B*	Disjunction, logical sum
$A \cdot B$	*A and B*	Conjunction, logical product

The use of $+$ for the *or* symbol arises from the parallelism between this logical operation and arithmetic addition. For example,

$$2 \text{ bananas} + 3 \text{ apples} = 5 \text{ pieces of fruit}$$

where pieces of fruit = bananas *or* apples. The parallelism between the logical product and the arithmetical product also holds; for example, 2 ft $\cdot$ 3 lb = 6 ft-lb since the unit of work, namely, the foot-pound, is a logical *and* combination.

Boolean Functions. Combinations of propositions joined by *or*, *not*, and *and* can be determined true or false and hence are propositions themselves. The uncombined propositions are called *elementary elements* and are usually denoted by letters at the beginning of the alphabet: A, B, ... Propositions combined by the propositional operations are called *combined elements*, or *Boolean functions*, and are usually denoted by letters at the end of the alphabet: X, Y, ... Combined elements are sometimes written in functional form, that is, $f(A,B, \ldots)$.

For example, consider the combined element $\bar{A} + B \cdot C$. Suppose that A is false, B is true, and C is false (that is, A might represent proposition a of the above examples, B might represent proposition b, and C might represent proposition e). In such a case $B \cdot C$ is false (since C is false), and $\bar{A}$ is true (since A is false); hence $\bar{A} + B \cdot C$ is true (since $\bar{A}$ is true). Note that $\bar{A} \cdot \bar{B}$ is distinct from $\overline{A \cdot B}$; the former is read (not A) and (not B), while the latter is read [not (A and B)] (the parentheses are for clarity). If A is true and B is false, then $\bar{A} \cdot \bar{B}$ is false and $\overline{A \cdot B}$ is true.

10-3. Boolean Expressions

There are two particularly important combined elements, namely, $\bar{A} + B$ and $A \cdot B + \bar{A} \cdot \bar{B}$. Their significance and meaning are discussed here.

Implication. First consider $\bar{A} + B$. This is usually abbreviated $A \rightarrow B$, called "*A implies B*," or "*If A, then B*," by logicians. In a sense this corresponds to the everyday linguistic use of the word *implies*, for if A is true and $A \rightarrow B$ is true, then B must be true; also, if $A \rightarrow B$ is true and B is true, A may or may not be true. For example, consider the proposition "if an airplane is a fighter plane, then it has light armor." Certainly, if the airplane is a fighter, it will have light armor, according to the proposition; but if the airplane has light armor, the proposition does not reveal whether it is a fighter or not, for other types of airplanes than fighters have light armor.

Equivalence. Consider now the important function $A \cdot B + \bar{A} \cdot \bar{B}$, called "*A is equivalent to B*" and abbreviated $A = B$. *If* $A \cdot B + \bar{A} \cdot \bar{B}$ *is true, then A and B have the same truth values,* for then either A and B are both true or they are both false. Also, the meaning of the equals ($=$) sign in logic parallels its meaning in arithmetic, in which any item can be substituted for an equivalent one. For it can be shown that, if $A = B$ is true, then some combination of the propositions U, V, and A, say $f(U,V,A)$, is equivalent to that combination with B replacing A, that is, $f(U,V,A) = f(U,V,B)$. In other words, it is always true that $(A = B) \rightarrow [f(U,V,A) = f(U,V,B)]$ in the propositional calculus.

Tautology. This naturally brings up the topic of tautology. A *tautology* is a combined element that is always true, independent of the truth or falseness of its component propositions. For example, $A + \bar{A}$, $(\overline{A \cdot B}) + A$, $(\overline{A \cdot B}) + (A + \bar{B}) \cdot (\bar{A} + B)$, as well as

$$(A = B) \rightarrow [f(U,V,A) = f(U,V,B)]$$

are all tautologies, for they are true no matter what the truth values of A, B, U, and V are (or what the truth value of the combined element f is). Therefore there is no logical reason to distinguish between these combined elements, and any one of them is called the universally true element, or tautology, and denoted by I. The negation of I, namely, $\bar{I}$, is the universally false element, denoted by 0. A consequence of these concepts is that $0 \rightarrow X$ and $X \rightarrow I$ are always true; i.e., the universally false element implies everything, while everything implies the tautology.

To review, then, there are two meanings to truth. One is "tautologically true," and the other is "true because it is a physical fact or supposition." "The grass is green" is true in the latter sense; "Today is Tuesday, or today is not Tuesday" is tautologically true. Logicians

often use a special symbol to distinguish these meanings, but for our limited purposes this would be an unnecessary complication.

Equations of Boolean Algebra. With the foregoing interpretation the algebraic properties of the logical operations can be more clearly understood. The following equations characterize a Boolean algebra. It is to be noted that these equations are quite different from the familiar rules of ordinary algebra. The subject that comprises algebraic manipulations according to the equations given below is therefore called *Boolean* algebra to distinguish it from ordinary algebra. Some of the equations of Boolean algebra are particularly important, and mathematicians have given them special names, as shown:

	Name of equation
1. $A + A = A$	Absorption rule for $+$
2. $A \cdot A = A$	Absorption rule for $\cdot$
3. $A + B = B + A$	Commutative rule for $+$
4. $A \cdot B = B \cdot A$	Commutative rule for $\cdot$
5. $(A + B) + C = A + (B + C) = A + B + C$	Associative rule for $+$
6. $(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C$	Associative rule for $\cdot$
7. $A \cdot (B + C) = A \cdot B + A \cdot C$	Distributive rule of $\cdot$ over $+$
8. $A + B \cdot C = (A + B) \cdot (A + C)$	Distributive rule of $+$ over $\cdot$
9. $\overline{A \cdot B} = \bar{A} + \bar{B}$	De Morgan's rule for $\cdot$
10. $\overline{A + B} = \bar{A} \cdot \bar{B}$	De Morgan's rule for $+$
11. $A + I = I$	
12. $A \cdot I = A$	
13. $0 + A = A$	
14. $0 \cdot A = 0$	
15. $\bar{A} + A = I$	
16. $A \cdot \bar{A} = 0$	
17. $A + A \cdot B = A$	
18. $A + \bar{A} \cdot B = A + B$	
19. $A \cdot B + B \cdot C + C \cdot A = (A + B) \cdot (B + C) \cdot (C + A)$	
20. $A \cdot \bar{B} + A \cdot C + B \cdot C = A \cdot \bar{B} + B \cdot C$	
21. $\bar{\bar{A}} = A$	

These equations are to be interpreted as tautologies. The equals sign means that, for any given combination of truth values for A, B, and C, the combined element on the right-hand side of the equation has the same truth value as the combined element on the left-hand side. Hence these equations can be checked by writing out truth values for the right-hand and left-hand combined elements.

Algebraic Proofs. Some of these equations can be "proved" algebraically by using others. For example, let us prove (20) by using (12), (15), (7), (6), (4), (3), and (17).†

† Note how difficult and rather "tricky" it is to prove (20). With experience and practice some facility can be gained in performing such manipulations. However, the student need not dwell on gaining such facility, for in Sec. 10-5 a systematic and straightforward method will be given for doing such problems.

$$
\begin{aligned}
& A \cdot \bar{B} + A \cdot C + B \cdot C \\
&= A \cdot \bar{B} + A \cdot C \cdot I + B \cdot C && \text{by (12)} \\
&= A \cdot \bar{B} + A \cdot C \cdot (\bar{B} + B) + B \cdot C && \text{by (15)} \\
&= A \cdot \bar{B} + (A \cdot C) \cdot \bar{B} + (A \cdot C) \cdot B + B \cdot C && \text{by (7)} \\
&= A \cdot \bar{B} + (A \cdot \bar{B}) \cdot C + B \cdot C + (B \cdot C) \cdot A && \text{by (6), (4), and (3)} \\
&= A \cdot \bar{B} + B \cdot C && \text{by (17) applied twice}
\end{aligned}
$$

New equations may also be proved by using those listed above. To show that $I \cdot A = A$, we have $I \cdot A = A \cdot I = A$ by (4) and (12).

There is another way of proving equations, namely, by making use of (15) and (11), for we want to show that an equation is a tautology (i.e., the meaning of "proving equations"). If we can show that the equation itself is equivalent to I (in truth value), then we have shown that it is a tautology, since I is the universally true element (i.e., always true for all truth-value combinations of the elementary elements).

For example, consider the proof of (17): $A + A \cdot B = A$ is an abbreviation for

$$(A + A \cdot B) \cdot A + (\overline{A + A \cdot B}) \cdot \bar{A}$$

but, on using (2), (3), (4), (6), (7), (9), (10), (11), and (15), we find that this expands to

$$
\begin{aligned}
A + A \cdot B + \bar{A} \cdot (\bar{A} + \bar{B}) \cdot \bar{A} &= A + A \cdot B + \bar{A} + \bar{A} \cdot \bar{B} \\
&= A + \bar{A} + A \cdot B + \bar{A} \cdot \bar{B} \\
&= I + A \cdot B + \bar{A} \cdot \bar{B} \qquad \text{by (15)} \\
&= I \qquad \text{by (11), to complete the proof}
\end{aligned}
$$

As another example, we can show that $A \cdot B \rightarrow A$ is a tautology, for $A \cdot B \rightarrow A$ is an abbreviation for $\overline{A \cdot B} + A$, which becomes, by (9), $\bar{A} + \bar{B} + A$. But $\bar{A} + \bar{B} + A = \bar{A} + A + \bar{B}$ [by (3) and (5)] = $I + \bar{B}$ [by (15)] = I [by (11)], to complete the proof.

There is still another way for proving equations, i.e., to make both sides of the equals sign look identical. For example, let us prove that $(A \rightarrow B) = (\bar{B} \rightarrow \bar{A})$. From the definition of $\rightarrow$ we have $\bar{A} + B = B + \bar{A}$ or $\bar{A} + B = \bar{A} + B$, by (21) and (3). As another example, let us show that

$$A \rightarrow (B \rightarrow C) = A \cdot B \rightarrow C$$

By the definition of $\rightarrow$ the left-hand side becomes $\bar{A} + \bar{B} + C$. Similarly the right-hand side becomes $\overline{A \cdot B} + C$; by using (9) this becomes $\bar{A} + \bar{B} + C$, as desired.

Other Logical Operations. It should be observed that Boolean algebra can be accomplished with just two operations, $+$ and $^-$, since the $\cdot$ operation can be given in terms of $+$ and $^-$ as $a \cdot b = \overline{\bar{a} + \bar{b}}$. There are in fact single operations in terms of which Boolean algebra can be accom-

plished. Consider the operation $p_1(a,b)$ defined in terms of $+$ and $^-$ as follows: $p_1(a,b) = \overline{a+b}$. To show that Boolean algebra can be accomplished in terms of this single operation, all we need do is define $+$, $\cdot$, and $^-$ in terms of $p_1(a,b)$. Thus, as can easily be checked from our definition, $\bar{a} = p_1(a,a)$; $a + b = p_1[p_1(a,b),p_1(a,b)]$; and $a \cdot b = p_1[p_1(a,a),p_1(b,b)]$. Another such single operation is $p_2(a,b) = \overline{a \cdot b}$. Here we have $\bar{a} = p_2(a,a)$, $a + b = p_2[p_2(a,a),p_2(b,b)]$, and $a \cdot b = p_2[p_2(a,b),p_2(a,b)]$. Note that I and 0 can be defined in terms of p_1 or p_2. For p_1 we have $I = p_1\{p_1[a,p_1(a,a)],p_1[a,p_1(a,a)]\}$, and then $0 = p_1(I,I)$. Can you define I and 0 in terms of p_2? If we admit I and 0 as primitive concepts, then there are four further single operations that can be defined, namely, $p_3(a,b) = \bar{a} + b$, $p_4(a,b) = a + \bar{b}$, $p_5(a,b) = \bar{a} \cdot b$, $p_6(a,b) = a \cdot \bar{b}$. For example, for p_3 we have $\bar{a} = p_3(a,0)$, $a + b = p_3[p_3(a,0),b]$, and $a \cdot b = p_3\{p_3[a,p_3(b,0)],0\}$. Can you show that p_4, p_5, and p_6 are each also valid single operations? What algebraic properties do $p_1, \ldots, p_6$ have? The single operation p_1 was first introduced by H. M. Sheffer.†

10-4. Representations of Boolean Expressions

Sets. We have introduced Boolean algebra as the algebra of propositional logic. However, Boolean algebra can also be interpreted as the algebra of sets, or classes. A *class* is a collection of objects all of which have a certain property; such a well-defined collection of objects is often called a *set*. For example, the collection of all automobiles manufactured in the year 1960 and still in operating condition is a set. Or the collection of people who registered at the national meeting of the Institute of Radio Engineers in 1960 forms a set. However, most often for the purposes of mathematics a set is visualized as the area inside a closed curve drawn on a piece of paper: the set is roughly interpreted as the collection of points that lie inside the closed curve (see Fig. 10-1). The objects that belong to the set are called *elements* of the set. For example, the point p is an element of the set (the interior of the circle) of Fig. 10-1. Sets are often denoted by letters; for example, we call the set in Fig. 10-1 the *set* A.

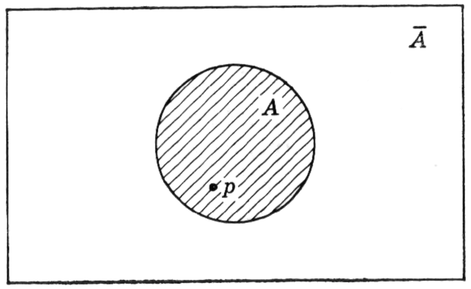


FIG. 10-1. Sets A and $\bar{A}$.

Just as the prime purpose of propositional logic was to determine the truth value of a proposition, so the prime purpose of the algebra of sets is

† A Set of Five Independent Postulates for Boolean Algebras, *Trans. Am. Math. Soc.*, vol. 14, pp. 481–488, 1913, where $p_1(a,b)$ was denoted by $a|b$, commonly known as *Sheffer's stroke*.

to determine *whether or not a point is in a set.* In fact the only difference between the propositional calculus of logic and the algebra of sets is that the algebra of sets deals with a very restricted kind of propositions. These are propositions of the form "*The point p is in the set A*" and combinations of such propositions with *and*, *or*, and *not*. Hence the hatched area shown in Fig. 10-1 represents all those points for which the proposition "The point p is in the set A" is true. Let us symbolize this proposition by A. Then $\bar{A}$ is true for all points p not lying in the hatched area.

Intersection and Union. All the points for which $A \cdot B$ is true (i.e., for which "the point p is in the set A, *and* the point p is in the set B" is true) are represented by the hatched area shown in Fig. 10-2. All the points for which $A + B$ is true (i.e., for which "the point p is in the set A, *or* the point p is in the set B" is true) are represented by the hatched area shown in Fig. 10-3. The hatched area shown in Fig. 10-2 is called the *intersection* of the set A with the set B, that is, the set of points that lie in both A *and* B. The hatched area shown in Fig. 10-3 is called the *union* of the set A with the set B, that is, the set of points that lie in either A *or* B or both.

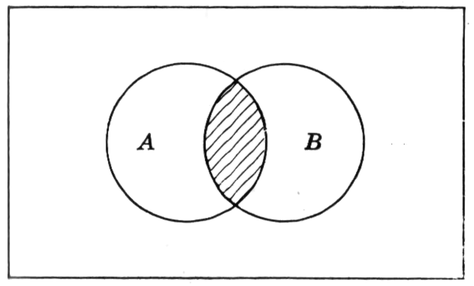


FIG. 10-2. Set $A \cdot B$.

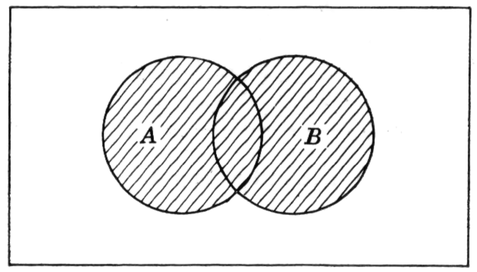


FIG. 10-3. Set $A + B$.

Now, as we have seen, Boolean algebra holds for propositions, and hence it must also hold for our set interpretation of propositions. Hence we can visualize some of the relations of Boolean algebra by means of the set diagrams. For example, consider the relation $\overline{A \cdot B} = \bar{A} + \bar{B}$. We shall show that $\overline{A \cdot B}$ and $\bar{A} + \bar{B}$ represent the same area. Figure 10-4 gives the area for $\bar{A}$, Fig. 10-5 for $\bar{B}$, and Fig. 10-6 for $\bar{A} + \bar{B}$. But, observing Fig. 10-2 for $A \cdot B$, we see that the area for $\bar{A} + \bar{B}$ is the same as that for $\overline{A \cdot B}$.

Just as in the propositional logic, given the truth value of the elementary elements, the truth value of a combined element can be determined, so, given the location of a point p with respect to the *elementary sets* $A, B, \ldots$, we can determine if it is in the set described by a combined element. For example, if a point p is in set A, not in set B, not in

set C, then is it in the set for which $(\bar{A} + \bar{B}) \cdot \bar{C}$ is true? The solution is given in Fig. 10-7. The hatched areas are all those points for which $(\bar{A} + \bar{B}) \cdot \bar{C}$ is true, and the point p is in set A, but not in set B or set C. Hence p is in the set $(\bar{A} + \bar{B}) \cdot \bar{C}$.

These diagrams representing sets by intersecting areas are called Venn diagrams (attributed to Venn by Schroeder). Figure 10-8 is such a diagram for four variables.

Logical Translation of English. The limitation on the practical application of the propositional calculus arises from the difficulty of translation of the practical worldly situations into these precise statements.

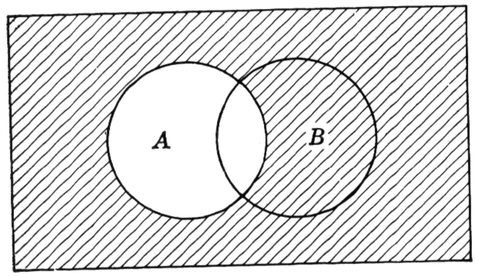


FIG. 10-4. Set $\bar{A}$.

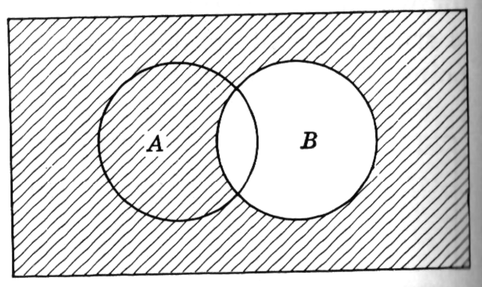


FIG. 10-5. Set $\bar{B}$.

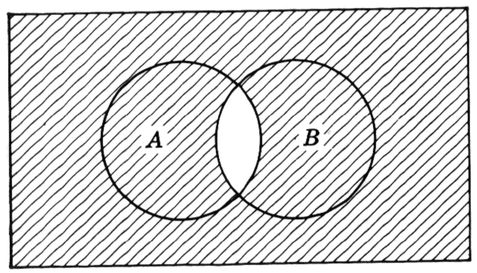


FIG. 10-6. Set $\bar{A} + \bar{B}$.

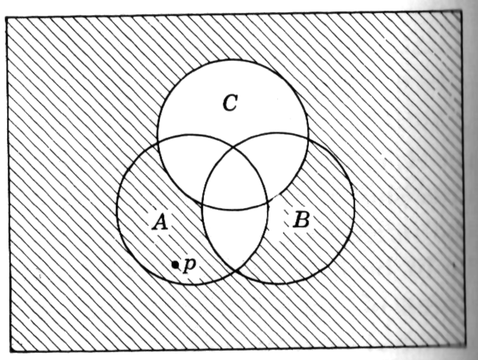


FIG. 10-7. Set $(\bar{A} + \bar{B}) \cdot \bar{C}$.

This translation, the first step in formulating the problem for solution, is by no means obvious and requires the work of a trained logical analyst. Part of this difficulty is the prevalent loose and often obscure use of language, and in particular of those very conjunctions which form the logical operations.

For example, the word *unless* as used in the combined proposition "A unless B" is defined by logicians as $A + B$, since "A unless B" is considered equivalent to "A if not B," that is, "if not B, then A," which is written symbolically as $\bar{B} \rightarrow A$. However, since not all users of the word *unless* are familiar with the logicians' delineation of the term, one often finds it employed with an entirely different connotation. For instance,

"A unless B" is often used to denote "If B, then not A," symbolically $B \rightarrow \bar{A}$, or $\bar{B} + \bar{A}$. A third possibility for "A unless B" might be "A, but if B then not A," which becomes $A \cdot (\bar{B} + \bar{A})$, or $A \cdot \bar{B}$. It is thus evident that in the formulation of the propositional elements from the

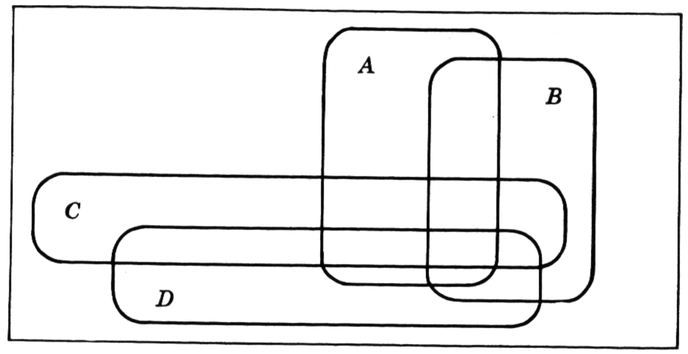


FIG. 10-8. Four-variable Venn diagram.

original raw sentences one must have enough familiarity with the subject matter to be able to discern the *intent* of a sentence in its given context.

In Table 10-1 we have suggested some translations for various English connectives. Caution must be used for the translation of *or* to distinguish

TABLE 10-1. TRANSLATIONS OF ENGLISH CONNECTIVES

English connective	*Logical translation*
Not A	$\bar{A}$
A and B	$A \cdot B$
A or (inclusive) B; A or B or both	$A + B$
A or (exclusive) B; either A or B	$A \neq B$, $A \cdot \bar{B} + \bar{A} \cdot B$
A but B	$A \cdot B$
A although B	$A \cdot B$
A unless B	$\bar{B} \rightarrow A$, $B + A$
A on condition that B	$B \rightarrow A$, $\bar{B} + A$
A if B	$B \rightarrow A$, $\bar{B} + A$
If A, then B; A implies B	$A \rightarrow B$, $\bar{A} + B$
A only if B†	$A \rightarrow B$, $\bar{A} + B$
Not unless A then B	$\bar{A} \rightarrow \bar{B}$, $A + \bar{B}$
A provided that B	$B \rightarrow A$, $\bar{B} + A$
A as well as B	$A \cdot B$
A if and only if B	$A = B$, $A \cdot B + \bar{A} \cdot \bar{B}$
Not both A and B	$\overline{A \cdot B}$, $\bar{A} + \bar{B}$
Neither A nor B	$\bar{A} \cdot \bar{B}$, $\overline{A + B}$
When A, then B	$A \rightarrow B$, $\bar{A} + B$
A because B	$B \rightarrow A$, $\bar{B} + A$

† Often imprecisely synonymous with "A if and only if B."

between the *inclusive or*, that is, *A or B or both*, and the *exclusive or*, that is, either *A or B* but *not* both. Also care must be observed in using "If A then B," or "A implies B." Strict logical interpretation is $A \rightarrow B$, or

$\bar{A} + B$. Consider, for example, $A \rightarrow B \cdot C$. This is equivalent to $(A \rightarrow B) \cdot (A \rightarrow C)$, which is reasonable in terms of the English translation. Similarly $A \rightarrow (B + C)$ is equivalent to $(A \rightarrow B) + (A \rightarrow C)$, which again is reasonable. However, $A \cdot B \rightarrow C$ is equivalent to $(A \rightarrow C) + (B \rightarrow C)$, which upon English translation is not obvious without considerable reflection; i.e., "Both A and B imply C" is the same as "A implies C, or B implies C, or both A implies C and B implies C." Similarly $(A + B) \rightarrow C$ is equivalent to $(A \rightarrow C) \cdot (B \rightarrow C)$, which again is not particularly obvious upon English translation.

Other Representations. A geometric representation of a Boolean algebra may be obtained by considering the vertices, edges, faces, etc., of a

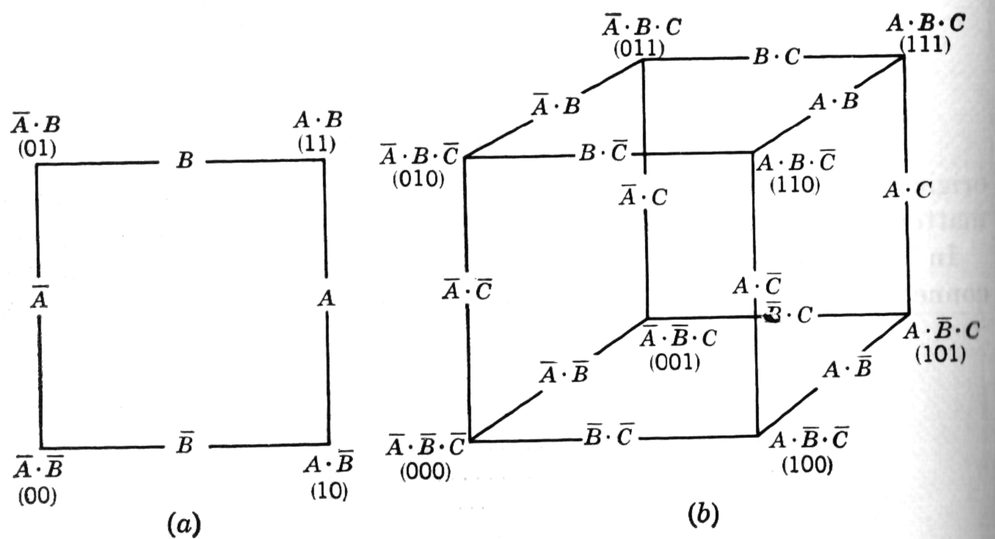


FIG. 10-9. Two- and three-dimensional hypercubes. The numbers represent basis columns (see Sec. 10-5).

hypercube, as in Fig. 10-9. For two variables we obtain a square (see Fig. 10-9*a*), where the vertices are the elementary products—that is, products of all the variables, barred or not (see below)—and the edges are the functions formed by the sum of the adjacent vertices. For three variables we obtain a cube, where now the faces are the sum of the adjacent vertices (or edges), and so forth (see Fig. 10-9*b*). In Fig. 10-9*b* we have labeled the edges; can you label the faces?

Another representation is a Boolean *lattice diagram*, illustrated in Fig. 10-10, for a two-elementary-element system. A point Y on the lattice is said to *include* another point X on the lattice if a continuous upward path can be found from X to Y along the lines drawn on the lattice diagram. For example, on our lattice, $\bar{A} + \bar{B}$ includes $A \cdot \bar{B}$; $A + B$ includes A, and so forth. One point of the lattice is said to be *below* (*above*) another point if it is on a row below (above) the row of the other point. For

example, on our lattice, $\bar{A} \cdot \bar{B}$ is below $\bar{B}$, $\bar{A} + \bar{B}$ is above $A \cdot \bar{B} + \bar{A} \cdot B$, etc.

The lattice is so constructed that the logical sum of two elements is the lowest point which *includes* both elements. For example, $\bar{A} + \bar{B}$ is the logical sum of $\bar{A}$ and $\bar{B}$, $A + B$ is the logical sum of $A \cdot \bar{B}$ and B, etc. The logical product of two elements is the highest point which *is included by* both elements. For example, $A \cdot B$ is the logical product of

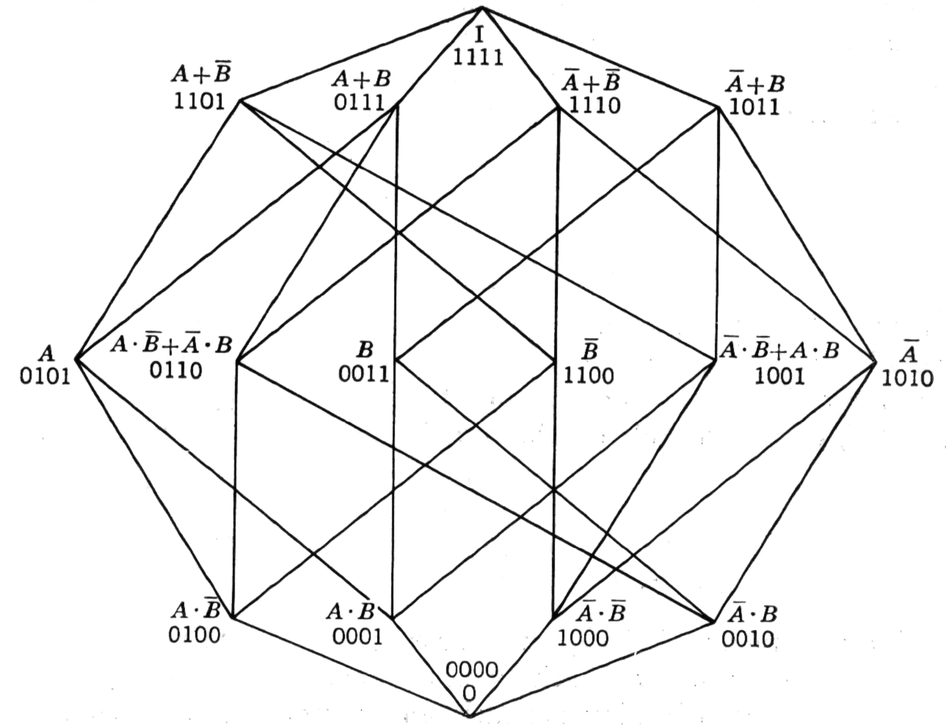


FIG. 10-10. Lattice diagram for two variables.

A and B; $A \cdot \bar{B}$ is the logical product of $A + B$ and $\bar{B}$. The numbers represent designation numbers (see below).

10-5. The Designation Numbers

The truth values of a combined element depend on the truth values of its component elementary elements, as dictated by the definitions of the propositional operations. A table can be made presenting all possible combinations of truth values; we call this a *truth table*. Consider, for example, $A + B$. The rows of A and B represent all possible truth-value combinations of A and B, and the rows of $A + B$ represent the corresponding truth values of $A + B$, where T stands for *true* and F for *false:*

A	F	T	F	T
B	F	F	T	T
$A + B$	F	T	T	T

Similarly the truth tables for $\bar{A}$ and $A \cdot B$ are

A	F	T
$\bar{A}$	T	F

A	F	T	F	T
B	F	F	T	T
$A \cdot B$	F	F	F	T

These truth tables follow from the definitions of the propositional operations and, conversely, could have been taken to be the definitions of these operations. With these basic truth tables in mind the truth table for any combined element can be constructed.

For example, recalling that $A \rightarrow B$ stands for $\bar{A} + B$ and that $A = B$ stands for $A \cdot B + \bar{A} \cdot \bar{B}$, the following table can be formed:

A	F	T	F	T
B	F	F	T	T
$A \rightarrow B$	T	F	T	T
$A = B$	T	F	F	T

Note that $A = B$ is true only when A and B have the same truth values. Similarly truth tables can be made for combinations of any number of elementary elements. For example, for three elementary elements A, B, and C, the truth table for the combined element $\bar{A} + B \cdot C$ becomes

$\bar{A}$	F	T	F	T	F	T	F	T
B	F	F	T	T	F	F	T	T
C	F	F	F	F	T	T	T	T
$\bar{A} + B \cdot C$	T	F	T	F	T	F	T	T

Using the truth-table idea, but replacing T by 1 and F by 0, we can form a binary number for each Boolean function. The computational methods to be described involve such binary numbers, called *designation numbers*. To every element, elementary or combined, we associate a designation number. In doing a problem, the designation-number representation for each proposition is obtained; the computations are performed in terms of these numbers; and finally the answer is reinterpreted in terms of propositions or sentences. The procedure for going from propositions to designation numbers will be considered first, and that for returning from designation numbers to propositions will follow.

The Basis. The designation numbers for the elementary elements† are assigned first. Such an assignment is called a basis. One such basis for a system of three elementary elements is‡

	0123	4567
$\#A$ =	0101	0101
$\#B$ =	0011	0011
$\#C$ =	0000	1111

† Also called *basic symbols* or *variables*.

‡ The # symbol means "The designation number of . . . ."

where the upper small numbers merely number the positions of the columns of the basis. For a system of n elementary elements, there are 2^n digits, or bits, in each designation number and hence 2^n columns in a basis. Any permutation of the columns of the basis shown will result in a different but nonetheless valid basis. The columns of a valid basis for n elementary elements need only represent all the 2^n possible combinations of 0 and 1 taken n at a time. Hence there can be $2^n!$ different bases.

One advantage of the type of basis illustrated above is that it may be written out directly for any number of elementary elements as follows: the number for the first elementary element alternates zeros and units; for the second alternates pairs of zeros and units; for the third alternates 4 zeros and 4 units; for the fourth would alternate 8 zeros and 8 units, and so forth, each such designation number being extended to comprise the full complement of 2^n bits. The column positions of the basis are numbered from the left, from 0 through $2^n - 1$. Such a basis is called a standard basis, and unless otherwise stated *it is always assumed that all designation numbers refer to this type of basis.* It is often convenient to denote this basis by $b[A,B,C]$, where the order of the elements in the square brackets is to be the same as the rows in the basis. Following this convention we have, for example, in the case of four elementary elements, the standard basis $b[A,B,C,D]$:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\#A$ =	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$\#B$ =	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$\#C$ =	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$\#D$ =	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Observe that each column of a standard basis, read from the bottom to the top, forms a binary number which is the same as the number of the column position. For example, column 3 is $\begin{smallmatrix}1\\1\\0\\0\end{smallmatrix}$ corresponding to 0011 = 3 (*decimal*), and column 14 is $\begin{smallmatrix}0\\1\\1\\1\end{smallmatrix}$ corresponding to 1110 = 14 (*decimal*).

Determining the Designation Number of a Boolean Function. In order to find $\#(A + B)$, we define an operation of "logical addition," to be performed on $\#A$ and $\#B$, wherein $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, and $1 + 1 = 1$ *without carry.* For example, with respect to the standard basis for two elementary elements,

$$\begin{array}{r} \#A = 0101 \\ \#B = 0011 \\ \hline \#(A + B) = 0111 \end{array}$$

To find $\#(A \cdot B)$, we similarly define the position-by-position operation of "logical multiplication," wherein $0 \cdot 0 = 0$, $0 \cdot 1 = 1 \cdot 0 = 0$, and

$1 \cdot 1 = 1$. For example,

$$\begin{aligned} \#A &= 0101 \\ \#B &= 0011 \\ \hline \#(A \cdot B) &= 0001 \end{aligned}$$

To find $\#\bar{A}$, we *invert* each digit of $\#A$, that is, change 0 to 1 and 1 to 0. For example,

$$\#\bar{A} = 1010$$

With these operations on designation numbers in mind, we determine the designation number of a combined element merely by performing the indicated operations with respect to the chosen basis. For example, to find $\#(\bar{A} + B \cdot C)$, we have, with respect to a basis for three elementary elements,

$$\begin{aligned} \#B &= 0011 \quad 0011 \\ \#C &= 0000 \quad 1111 \\ \hline \#(B \cdot C) &= 0000 \quad 0011 \\ \#\bar{A} &= 1010 \quad 1010 \\ \hline \#(\bar{A} + B \cdot C) &= 1010 \quad 1011 \end{aligned}$$

There should be no confusion as to the dual role of the symbols $+$, $\cdot$, $^-$. When applied to propositions A, B, C, . . . , they are the logical operations *or*, *and*, and *not*, respectively; when applied to designation numbers $\#A$, $\#B$, $\#C$, . . . , they are the numerical operations *logical add*, *logical multiply*, and *invert*, respectively.

Important Results. Some important results of this digitalization of propositions are: (1) $\#I = 1111 \cdots$; that is, $\#I$ has all units. (2) $\#X = \#Y$ if, and only if, $X = Y$. (3) $X \rightarrow Y$ if, and only if, $\#Y$ has units in at least those positions which correspond to the units of $\#X$.

For example, to demonstrate equivalences, we simply calculate the designation number of each side; if they are identical, the equation holds. To show that $A \cdot B + A \cdot C + \bar{B} \cdot C = A \cdot B + \bar{B} \cdot C$, we find

$$\#(A \cdot B) = 0001 \quad 0001 \qquad \#(A \cdot C) = 0000 \quad 0101$$

and

$$\#(\bar{B} \cdot C) = 0000 \quad 1100$$

whence

$$\#(A \cdot B + A \cdot C + \bar{B} \cdot C) = 0001 \quad 1101$$

but $\#(A \cdot B + \bar{B} \cdot C) = 0001\ 1101$ also, whence the equivalence must hold. To show that an implication is true, simply show that the implied proposition has units in at least the same position as the implier. For example, to demonstrate that $A \cdot B + \bar{B} \cdot C \rightarrow A + C$, we note that

$$\#(A \cdot B + \bar{B} \cdot C) = 0001\ \overset{3\ 4\ 5\ 7}{1101},$$

which has units in positions 3, 4, 5, and 7; also $\#(A + C) = 0101\ 1111$, which obviously has units in at least these positions.

10-6. Forms of Boolean Functions

From Designation Number to Boolean Function. Given a designation number, there are systematic methods for finding its symbolic Boolean algebraic representation in several different forms. For an example of five different equivalent representations of the same designation number, consider the number 0111 0100. (1) The first canonical form of this is $A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C$. (2) The second canonical form is $(A + B + C) \cdot (A + B + \bar{C}) \cdot (A + \bar{B} + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C})$. (3) The simplest sum-of-products form is $A \cdot \bar{B} + B \cdot \bar{C}$. (4) The simplest product-of-sums form is $(A + B) \cdot (\bar{B} + \bar{C})$. (5) A mongrel form is $(A + B) \cdot \bar{C} + A \cdot \bar{B} \cdot C$. All these combined elements are equivalent. In this section systematic methods for obtaining the first two and last forms are given; methods for finding the third form are given in the following section; the simplest product-of-sums form is left as an exercise.

The First Canonical Form. The so-called *first canonical form*, also called the disjunctive normal form, is a sum of elementary products. An *elementary product* is a product that contains *all* the elementary elements or their negations. For example, all possible elementary products for three elementary elements are $\bar{A} \cdot \bar{B} \cdot \bar{C}$, $A \cdot \bar{B} \cdot \bar{C}$, $\bar{A} \cdot B \cdot \bar{C}$, $A \cdot B \cdot \bar{C}$, $\bar{A} \cdot \bar{B} \cdot C$, $A \cdot \bar{B} \cdot C$, $\bar{A} \cdot B \cdot C$, and $A \cdot B \cdot C$. Observe that the designation number of an elementary product always has a single unit: e.g.,

$$\begin{aligned} \#(\bar{A} \cdot \bar{B} \cdot \bar{C}) &= 1000 \quad 0000 \\ \#(A \cdot \bar{B} \cdot \bar{C}) &= 0100 \quad 0000 \\ \#(\bar{A} \cdot B \cdot \bar{C}) &= 0010 \quad 0000 \\ \#(A \cdot B \cdot \bar{C}) &= 0001 \quad 0000 \\ \#(\bar{A} \cdot \bar{B} \cdot C) &= 0000 \quad 1000 \\ \#(A \cdot \bar{B} \cdot C) &= 0000 \quad 0100 \\ \#(\bar{A} \cdot B \cdot C) &= 0000 \quad 0010 \\ \#(A \cdot B \cdot C) &= 0000 \quad 0001 \end{aligned}$$

For n elementary elements there are 2^n such products. The first canonical form of a designation number is the sum of only those elementary products each of which contributes a unit to the number. For example, 0100 0011 has units in positions 1, 6, and 7 and can therefore be constructed by summing the elementary products whose designation numbers have units in positions 1, 6, and 7. Hence, we have

$$\begin{aligned} \#(A \cdot \bar{B} \cdot \bar{C}) &= 0100 \quad 0000 \\ \#(\bar{A} \cdot B \cdot C) &= 0000 \quad 0010 \\ \#(A \cdot B \cdot C) &= 0000 \quad 0001 \\ \hline \#(A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot C + A \cdot B \cdot C) &= 0100 \quad 0011 \end{aligned}$$

i.e., the first-canonical-form representation of 0100 0011 is

$$A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot C + A \cdot B \cdot C$$

The Second Canonical Form. To find the *second canonical form,* also called the conjunctive normal form, note that the designation number of the *sum* of all the elementary elements or their negations has exactly one zero:

$$\begin{aligned}
\#(\bar{A} + \bar{B} + \bar{C}) &= 1111 \quad 1110 \\
\#(A + \bar{B} + \bar{C}) &= 1111 \quad 1101 \\
\#(\bar{A} + B + \bar{C}) &= 1111 \quad 1011 \\
\#(A + B + \bar{C}) &= 1111 \quad 0111 \\
\#(\bar{A} + \bar{B} + C) &= 1110 \quad 1111 \\
\#(A + \bar{B} + C) &= 1101 \quad 1111 \\
\#(\bar{A} + B + C) &= 1011 \quad 1111 \\
\#(A + B + C) &= 0111 \quad 1111
\end{aligned}$$

These sums are called *elementary sums;* for n elements there are 2^n elementary sums. The conjunctive normal form corresponding to a given designation number is the product of those elementary sums which correspond to the zeros in the designation number. For example, for
0 2 3 4 5
0100 0011 we have zeros in positions 0, 2, 3, 4, and 5, and hence

$$\#[(A + B + C) \cdot (A + \bar{B} + C) \cdot (\bar{A} + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + B + \bar{C})] = 0100 \quad 0011$$

This follows because each elementary sum in the product contributes one of the zeros of the designation number.

$$\begin{aligned}
\#(A + B + C) &= 0111 \quad 1111 \\
\#(A + \bar{B} + C) &= 1101 \quad 1111 \\
\#(\bar{A} + \bar{B} + C) &= 1110 \quad 1111 \\
\#(A + B + \bar{C}) &= 1111 \quad 0111 \\
\#(\bar{A} + B + \bar{C}) &= 1111 \quad 1011 \\
\hline
\#\text{(product)} &= 0100 \quad 0011
\end{aligned}$$

as desired.

Mongrel Forms. The *mongrel form* is not a unique form; i.e., two people determining a mongrel form for the same designation number may not arrive at the same Boolean function. In general, in the mongrel form anything goes, just as long as the final Boolean function actually represents the designation number under consideration. For example, consider the designation number 0001 0111. We note that

$$\begin{aligned}
\#A &= 0101 \quad 0101 \\
\#B &= 0011 \quad 0011 \\
\#C &= 0000 \quad 1111 \\
\hline
\#(A \cdot B) &= 0001 \quad 0001 \\
\#(A + B) &= 0111 \quad 0111
\end{aligned}$$

But since $\#\bar{C} = 1111\ 0000$, then $\#(A \cdot B \cdot \bar{C}) = 0001\ 0000$; also, since $\#C = 0000\ 1111$, then $\#[(A + B) \cdot C] = 0000\ 0111$, whence

$$0001 \quad 0111 = \#[A \cdot B \cdot \bar{C} + (A + B) \cdot C]$$

or $A \cdot B \cdot \bar{C} + (A + B) \cdot C$ is a mongrel form. The mongrel form is particularly useful in more complicated cases. For example, consider 0001 0111 0111 0001 1110 1110 1000 1000. Here we have the basis

$$\begin{aligned}
\#A &= 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101 \\
\#B &= 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011 \\
\#C &= 0000\ 1111\ 0000\ 1111\ 0000\ 1111\ 0000\ 1111 \\
\#D &= 0000\ 0000\ 1111\ 1111\ 0000\ 0000\ 1111\ 1111 \\
\#E &= 0000\ 0000\ 0000\ 0000\ 1111\ 1111\ 1111\ 1111
\end{aligned}$$

Since

$$\begin{aligned}
\#(A \cdot B) &= 0001\ 0001\ 0001\ 0001\ 0001\ 0001\ 0001\ 0001 \\
\#(A + B) &= 0111\ 0111\ 0111\ 0111\ 0111\ 0111\ 0111\ 0111 \\
\#(\bar{A} + \bar{B}) &= 1110\ 1110\ 1110\ 1110\ 1110\ 1110\ 1110\ 1110 \\
\#\bar{A} \cdot \bar{B} &= 1000\ 1000\ 1000\ 1000\ 1000\ 1000\ 1000\ 1000
\end{aligned}$$

and

$$\begin{aligned}
\#(\bar{C} \cdot \bar{D} \cdot \bar{E}) &= 1111\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \\
\#(C \cdot \bar{D} \cdot \bar{E}) &= 0000\ 1111\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \\
\#(\bar{C} \cdot D \cdot \bar{E}) &= 0000\ 0000\ 1111\ 0000\ 0000\ 0000\ 0000\ 0000 \\
\#(C \cdot D \cdot \bar{E}) &= 0000\ 0000\ 0000\ 1111\ 0000\ 0000\ 0000\ 0000 \\
\#(\bar{D} \cdot E) &= 0000\ 0000\ 0000\ 0000\ 1111\ 1111\ 0000\ 0000 \\
\#(D \cdot E) &= 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1111\ 1111
\end{aligned}$$

we can immediately write down the following mongrel form:

$$(A \cdot B) \cdot (\bar{C} \cdot \bar{D} \cdot \bar{E}) + (A + B) \cdot (C \cdot \bar{D} \cdot \bar{E}) + (A + B) \cdot (\bar{C} \cdot D \cdot \bar{E}) + (A \cdot B) \cdot (C \cdot D \cdot \bar{E}) + (\bar{A} + \bar{B}) \cdot (\bar{D} \cdot E) + (\bar{A} \cdot \bar{B}) \cdot (D \cdot E)$$

A useful fact to remember in writing mongrel forms is that

$$\begin{aligned}
0110 &= \#(A \cdot \bar{B} + \bar{A} \cdot B) \\
\text{and} \qquad 1001 &= \#(A \cdot B + \bar{A} \cdot \bar{B})
\end{aligned}$$

Of course this can be generalized to the observations, for example, that

$$0000\ 1111\ 1111\ 0000\ 0000\ 1111\ 1111\ 0000 = \#(C \cdot \bar{D} + \bar{C} \cdot D)$$

and

$$1111\ 0000\ 0000\ 1111\ 1111\ 0000\ 0000\ 1111 = \#(C \cdot D + \bar{C} \cdot \bar{D})$$

Applying this to 0001 0111 0111 0001 1110 1110 1000 1000, we find the mongrel form

$$[(A \cdot B) \cdot (C \cdot D + \bar{C} \cdot \bar{D}) + (A + B) \cdot (C \cdot \bar{D} + \bar{C} \cdot D)] \cdot \bar{E} + [(\bar{A} + \bar{B}) \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot D] \cdot E$$

The writing of "good" mongrel forms depends on ingenuity and practice. Of course the individual component products and sums need not be written out explicitly as was done above. For example, consider 0110 1001 1001 0110. A little thought will enable the following mongrel form to be written down immediately:

$$(A \cdot \bar{B} + \bar{A} \cdot B) \cdot (C \cdot D + \bar{C} \cdot \bar{D}) + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot (C \cdot \bar{D} + \bar{C} \cdot D)$$

10-7. Simplest Sum-of-products Representation†

The Role of the Prime Implicants. This symbolic representation of a designation number has the property of being a *sum of products with the least number of operations · and +*. There may be several such forms associated with a single designation number. A systematic process can be described that will result in such a form. To understand this process, let us first discuss the general nature of a product.

The number of letters that enter into a product is called the number of *terms* in the product. Thus $A \cdot \bar{B} \cdot C$ is a three-term product, $A \cdot \bar{B}$ is a two-term product, and A is called a one-term product. In general, for n basic symbols, the designation number of an n-term product has 2^0 $(=1)$ unit, of an $(n-1)$-term product the designation number has 2^1 units, of an $(n-2)$-term product it has 2^2 units, . . . , of an $(n-s)$-term product the designation number has 2^s units. For example, $\#(A \cdot \bar{B} \cdot C) = 0000\ 0100$ with 1 unit, $\#(A \cdot \bar{B}) = 0100\ 0100$ with 2 units, and $\#A = 0101\ 0101$ with 4 units. Another observation about products that follows directly is that, if P and P' are products such that $\#P \rightarrow \#P'$, then P' is formed from P by deleting some of the terms of P. For example, $\#(A \cdot \bar{B} \cdot C) \rightarrow \#(A \cdot \bar{B})$, the latter being formed by deleting C from the former. The converse is evidently also true.

Now we can turn to the discussion of the simplest sum-of-products form. If $\#F$ is the designation number in question, consider a simplest sum-of-products representation of $\#F$. Clearly each product P_i of the sum must be such that $\#P_i \rightarrow \#F$ (that is, $\#P_i$ cannot have units in positions where $\#F$ has zeros), for otherwise the sum of the products would not have the same designation number as $\#F$. Also, if this sum is indeed the *simplest* sum of products, then certainly no product P_i of this sum can be replaced by a product P_i' such that $P_i \rightarrow P_i'$, for then the resulting sum of products would become even simpler (i.e., one of the products would have a term deleted). In other words, *in order that a product P be considered at all for inclusion in a simplest sum-of-products form for $\#F$, it must be such that:*

1. $\#P \rightarrow \#F$.
2. *There exists no P' such that $\#P \rightarrow \#P'$ and $\#P' \rightarrow \#F$.*

If P has the first property, it is called an *implicant* of F; if it also has the second, it is a *prime implicant* of F.

† The method of this section is an adaptation of McCluskey's variation of Quine's method of simplification. (See E. J. McCluskey, Jr., Minimization of Boolean Functions, *Bell System Tech. J.*, vol. 35, p. 1417, November, 1956. See also W. v. O. Quine, The Problem of Simplifying Truth Functions, *Am. Math. Monthly*, vol. 59, no. 8, pp. 521–531, October, 1952.)

For example if $\#F = 0111\ 0100$, then $A \cdot \bar{B} \cdot \bar{C}$, $\bar{A} \cdot B \cdot \bar{C}$, $A \cdot B \cdot \bar{C}$, $A \cdot \bar{B} \cdot C$, $A \cdot \bar{C}$, $A \cdot \bar{B}$, and $B \cdot \bar{C}$ are all implicants of F, as can be readily checked, but only $A \cdot \bar{C}$, $A \cdot \bar{B}$, and $B \cdot \bar{C}$ are *prime implicants* of F. To show that $A \cdot \bar{B}$ is a prime implicant of F, note that the only products implied by $A \cdot \bar{B}$ are A and $\bar{B}$. But $\#A \nrightarrow \#F$, and $\#\bar{B} \nrightarrow \#F$, and hence $A \cdot \bar{B}$ is a prime implicant of F. (On the other hand, $A \cdot \bar{B} \cdot C$ is not a prime implicant because $\#A \cdot \bar{B} \cdot C \rightarrow \#A \cdot \bar{B}$ and $\#A \cdot \bar{B} \rightarrow \#F$.)

There are two stages in determining the simplest sum-of-products form of F. The first is the systematic generation of *all* the *prime* implicants of F; the *second* is the selection from this collection of a subcollection of products the sum of which will truly be the simplest sum of products, as defined above. We shall now describe in detail how to generate all the prime implicants of a given designation number. Then we shall show how with the aid of a so-called *prime-implicant chart* the final simplest sum of products is chosen.

The First Stage—Generating the Prime Implicants. The rationale for generating the prime implicants can best be understood by examining the first canonical form. For 0111 0100 the first canonical form is $A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C$. The following reductions can be made by combining pairs of products:

$$A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot C = A \cdot \bar{B} \cdot (\bar{C} + C) = A \cdot \bar{B} \cdot I = A \cdot \bar{B}$$
$$\bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} = (\bar{A} + A) \cdot B \cdot \bar{C} = I \cdot B \cdot \bar{C} = B \cdot \bar{C}$$

Thus the first canonical form reduces to $A \cdot \bar{B} + B \cdot \bar{C}$. We can determine whether or not two products can be combined by comparing the pair of associated basis columns: if the two columns differ in just one row, the terms can be combined. For instance, the pair $A \cdot \bar{B} \cdot \bar{C}$, $A \cdot \bar{B} \cdot C$ corresponds to the columns $\begin{smallmatrix}1\\0\end{smallmatrix}$, $\begin{smallmatrix}1\\0\end{smallmatrix}$, which differ only in the third row; similarly the pair $\bar{A} \cdot B \cdot \bar{C}$, $A \cdot B \cdot \bar{C}$ corresponds to the columns $\begin{smallmatrix}0\\1\end{smallmatrix}$, $\begin{smallmatrix}1\\0\end{smallmatrix}$, which differ only in the first row. Based on these concepts, the first stage involves three steps. We shall illustrate these steps by following through a specific example.

Consider the following designation number in four basic symbols:

0	1		3		4		6	7		9	10	11		13	14	15
1	1	0	1		1	0	1	1		0	1	1	1		0	1	1	1

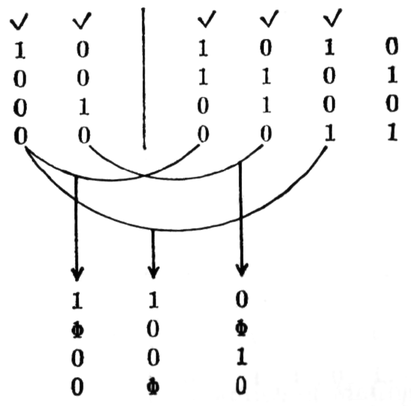
The first step is to display only the columns of the basis that correspond to the *units* of the given designation number:

0	1	3	4	6	7	9	10	11	13	14	15
0	1	1	0	0	1	1	0	1	1	0	1
0	0	1	0	1	1	0	1	1	0	1	1
0	0	0	1	1	1	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1

The second step is to arrange these columns in order by the number of units contained in each column:

0-unit columns	1-unit columns	2-unit columns	3-unit columns	4-unit columns
0	10	1010	1110	1
0	00	1101	1101	1
0	01	0100	1011	1
0	00	0011	0111	1

The third step is an iterative process of combining pairs of columns that differ only in a single row. For example, $\begin{smallmatrix}0\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ are combined to form $\begin{smallmatrix}\Phi\\0\\0\\0\end{smallmatrix}$, where the symbol Φ means 0 or 1. Similarly $\begin{smallmatrix}0\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}0\\0\\1\\0\end{smallmatrix}$ are combined to form $\begin{smallmatrix}0\\0\\\Phi\\0\end{smallmatrix}$. Note that, since the pairs are to differ in a single row only, each u-unit column need be compared with only $(u + 1)$-unit columns. To *combine* with a u-unit column, a $(u + 1)$-unit column must have u of its $u + 1$ units in the same rows as the u-unit column [i.e., the u-unit column must imply the $(u + 1)$-unit column]. Thus the 1-unit columns are combined with the 2-unit columns as follows:



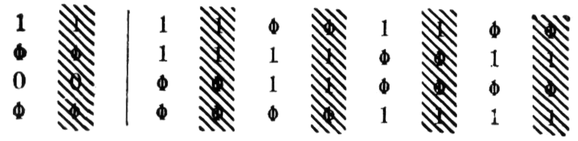
When a pair of columns are combined, a check is placed above them to record the fact. As will be seen shortly, this checking process is of the utmost importance. In this way the adjacent collections of u-unit

columns and $(u + 1)$-unit columns are successively compared and checked. In our example, this results in

✓	✓✓	✓✓✓✓	✓✓✓✓	✓
0	10	1010	1110	1
0	00	1101	1101	1
0	01	0100	1011	1
0	00	0011	0111	1
Φ0	10	11Φ011Φ0	111Φ	
00	Φ0Φ	1111Φ011	11Φ1	
0Φ	001	Φ0110Φ0Φ	1Φ11	
00	0Φ0	0Φ0Φ1111	Φ111	

The process is continued on this and each successive set of combined columns. But this time for two combined columns to be further combined they must have Φ's as well as units in the same rows. The process ends when no further combinations can be formed. For our example the rest of the work is shown below:

	✓✓	✓✓✓✓✓✓✓✓	✓✓✓✓
Φ0	10	11Φ011Φ0	111Φ
00	Φ0Φ	1111Φ011	11Φ1
0Φ	001	Φ0110Φ0Φ	1Φ11
00	0Φ0	0Φ0Φ1111	Φ111



Duplicate columns have been shown hatched above. The prime implicants can all now be read from the *unchecked* columns. For example, $\begin{smallmatrix}\Phi\\0\\0\\0\end{smallmatrix}$ corresponds to $\bar{B} \cdot \bar{C} \cdot \bar{D}$, or $\begin{smallmatrix}1\\\Phi\\0\\\Phi\end{smallmatrix}$ corresponds to $A \cdot \bar{C}$, and so forth. For our example the prime implicants are thus $\bar{B} \cdot \bar{C} \cdot \bar{D}$, $\bar{A} \cdot \bar{B} \cdot \bar{D}$, $\bar{A} \cdot C \cdot \bar{D}$, $A \cdot \bar{C}$, $A \cdot B$, $B \cdot C$, $A \cdot D$, and $B \cdot D$.

The Second Stage—Using the Prime-implicant Chart. A chart showing the prime implicants is used in selecting for inclusion in the simplest sum of products those prime implicants which will produce the given designation number with the least number of operations · and +. The chart is simply a display of the prime-implicant designation-number unit positions. For our example this chart is shown in Table 10-2. (Note that, to determine, for instance, the unit positions of $\#\bar{B} \cdot \bar{C} \cdot \bar{D}$, we simply

observe that $\bar{B} \cdot \bar{C} \cdot \bar{D}$ corresponded to $\begin{smallmatrix}\Phi\\0\\0\\0\end{smallmatrix}$. The Φ in the first row means that A was eliminated between the *two* columns $\begin{smallmatrix}0\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$; the zeros indicate that B, C, and D must be complemented in the three positions. As another illustration, $A \cdot \bar{C}$ corresponds to $\begin{smallmatrix}1\\\Phi\\0\\\Phi\end{smallmatrix}$; the Φ's indicate that B and D have been eliminated, and the zero indicates that C must be complemented.)

TABLE 10-2. PRIME-IMPLICANT CHART

	Unit positions of the given designation number $\#F$											
	0	1	3	4	6	7	9	10	11	13	14	15
$\bar{B} \cdot \bar{C} \cdot \bar{D}$	0	1										
$\bar{A} \cdot \bar{B} \cdot \bar{D}$	0			4								
$\bar{A} \cdot C \cdot \bar{D}$				4	6							
$A \cdot \bar{C}$		1	3				9		11			
$A \cdot B$			3			7			11			15
$B \cdot C$					6	7					14	15
$\longrightarrow A \cdot D$							9		11	13		15
$\longrightarrow B \cdot D$								10	11		14	15

To choose the appropriate prime implicants for inclusion in the simplest sum of products, *first* note that $A \cdot D$ and $B \cdot D$ *must* be included because the units of $\#F$ in positions 10 and 13 are covered only by $B \cdot D$ and $A \cdot D$, respectively. If either $B \cdot D$ or $A \cdot D$ were left out of our sum, the designation number of the sum, lacking units in positions 10 or 13, could never be the same as $\#F$. Prime implicants that must for this reason be included are called *essential prime implicants.* Thus part of our sum must be $A \cdot D + B \cdot D$.

Next note that $\#(A \cdot D + B \cdot D)$ covers positions 9, 10, 11, 13, 14, and 15, leaving only positions 0, 1, 3, 4, 6, and 7 to be covered by additional prime implicants. The *second* step in using the prime-implicant chart is, then, to choose in the best possible way from the *nonessential prime implicants* those combinations whose sum will cover unit positions of $\#F$ not already covered by the sum of the essential prime implicants. There is no known systematic procedure for doing this, except for trying all combinations. For our illustration it can be seen that $\bar{A} \cdot \bar{B} \cdot \bar{D} + A \cdot \bar{C} + B \cdot C$ will cover positions 0, 1, 3, 4, 6, and 7 in the best way. (Note that $B \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot C \cdot \bar{D} + A \cdot B$ will also cover positions 0, 1, 3, 4, 6, and 7, but has one more operation · than the best choice.)

Thus we finally have for the simplest sum-of-products form of the designation number 1101 1011 0111 0111

$$A \cdot D + B \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{D} + A \cdot \bar{C} + B \cdot C$$

Summary. Summarizing, the process of finding the simplest sum-of-products form for a given designation number $\#F$ has two stages. The first stage is the determination of all the prime implicants of $\#F$ by combining pairs of basis columns that correspond to the unit positions of $\#F$. To aid the formation of the combinations, the columns are first ordered into a 0-unit collection, 1-unit collection, 2-unit collection, etc. The prime implicants are represented by those columns which do not enter into any further combinations, i.e., which are left unchecked when all combinations have been made. The second stage is to choose in the best way those prime implicants which are to be included in the simplest sum of products. As an aid to making this choice the prime-implicant chart is constructed, and from this essential prime implicants are noted. Then from the nonessential prime implicants a best sum is chosen to cover those units of $\#F$ not covered by the sum of the essential prime implicants. The simplest sum of products then becomes the sum of the essential prime implicants and this best sum of the nonessential prime implicants.

Decimal Method for Simplification.† The work of finding the simplest sum-of-products form by the method just given can be shortened by considering the decimal numbers denoting basis columns instead of the columns themselves. Consider again the example of this section, the designation number

	0	1		3		4		6	7		9	10	11		13	14	15
$\#F =$	1	1	0	1		1	0	1	1		0	1	1		0	1	1	1

Instead of writing the columns in groups of ascending numbers of included units,

$$\begin{array}{c|cc|cc} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{array} \quad \cdots$$

we merely write the groups of decimal numbers of those columns: 0; 1, 4; 3, 6, 9, 10; 7, 11, 13, 14; 15. Note that if a column is to combine with another in the next adjacent group then the number in the latter group must be *greater* than the number in the former by a power of 2. Hence we need list (in groups) for the combined result *only the least number, and the difference,* which corresponds to the power of 2 by which the columns differ: 0-1 for columns 0 and 1, 0-4 for columns 0 and 4, 1-2 for 1 and 3, 1-8 for 1 and 9, 4-2 for 4 and 6 (but *not* 4-1 for 4 and 3), 3-4 for 3 and 7, etc. Just as combined columns recombine only if they have Φ

† This method was developed by James Bruce Wilson.

in the same positions, so these pairs combine only if they have common differences: thus 1-2 and 9-2 combine, and we write 1-2,8 (also 1-8 and 3-8 combine to form 1-2,8). The work for this example would be tabulated as in Table 10-3. No combinations are possible in the final row of the table, because no pairs have the same two differences.

The prime implicants are represented by the unchecked numbers of Table 10-3: namely, 0-1; 0-4; 4-2; 1-2,8; 3-4,8; 6-1,8; 9-2,4; 10-1,4. The

TABLE 10-3. WORK FOR THE DECIMAL METHOD

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

0 | 1; 4 | 3; 6; 9; 10 | 7; 11; 13; 14 | 15

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

0-1; 0-4 | 1-2; 1-8; 4-2 | 3-4; 3-8; 6-1; 6-8; 9-2; 9-4; 10-1; 10-4 | 7-8; 11-4; 13-2; 14-1

1-2,8; 1-8,2 | 3-4,8; 6-1,8; 6-8,1; 9-2,4; 9-4,2; 10-1,4; 10-4,1

or, on eliminating duplications,

1-2,8 | 3-4,8; 6-1,8; 9-2,4; 10-1,4

prime-implicant chart (see Table 10-2) is formed by adding to each least number all combinations of its associated differences. To find the expression for one of these combination numbers, we omit variables that correspond to powers of 2 found in the differences and bar those variables whose powers do not add to make the first number, to the left of the dash. Thus 9-2,4 corresponds to $A \cdot D$ (since B and C are deleted and $1 + 8 = 9$) and 4-2 corresponds to $\bar{A} \cdot C \cdot \bar{D}$ (since B is deleted and $0 + 4 + 0 = 4$).

10-8. Obtaining Essential Prime Implicants Directly†

Preliminary Notational Review. Let us recall the meaning of our Φ notation. If $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ is column 1 of a basis, $\begin{smallmatrix}1\\0\\0\\\Phi\end{smallmatrix}$ represents both $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}1\\0\\0\\1\end{smallmatrix}$. Similarly $\begin{smallmatrix}1\\\Phi\\\Phi\\\Phi\end{smallmatrix}$ represents the 2^3 basis columns $\begin{smallmatrix}1&1&1&1&1&1&1&1\\0&0&0&1&0&1&1&1\\0,&0,&1,&0,&1,&0,&1,&1.\\0&1&0&0&1&1&0&1\end{smallmatrix}$ If a column has Φ in p of its rows, it represents 2^p basis columns, in which the p rows are filled in all 2^p possible ways. Recall also that we interpret $\begin{smallmatrix}1\\0\\0\\\Phi\end{smallmatrix}$ to mean $A \cdot \bar{B} \cdot \bar{C}$, with D eliminated; similarly $\begin{smallmatrix}1\\\Phi\\\Phi\\\Phi\end{smallmatrix}$ means A, with B, C, and D eliminated, and so forth. We interpret a basis column as a binary number, with the least significant digit on top, and thus the rows in a

† The method of this section is based on R. D. Elbourn's adaptation of a method developed by Harris. (See B. Harris, An Algorithm for Determining Minimal Representation of a Logic Function, *IRE Trans. on Electronic Computers*, vol. EC-6, pp. 103–108, June, 1957.)

basis column are associated with successive powers of 2; for example, for four variables

$$A \leftrightarrow 2^0 = 1$$
$$B \leftrightarrow 2^1 = 2$$
$$C \leftrightarrow 2^2 = 4$$
$$D \leftrightarrow 2^3 = 8$$

Hence we shall refer to the *power position* of a bit in a column (e.g., column 1, namely, $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$, has a unit in power position 1, and $\begin{smallmatrix}1\\0\\0\\1\end{smallmatrix}$ has units in power positions 1 and 8). Finally, observe that the 2^p basis column numbers represented by a column with p of its components Φ can be found by taking the number formed with the Φ's read as zeros and adding all combinations of powers of 2 corresponding to the power positions of the Φ's. For example, $\begin{smallmatrix}1\\0\\0\\\Phi\end{smallmatrix}$ represents basis columns 1 and $1 + 8 = 9$, and $\begin{smallmatrix}1\\\Phi\\\Phi\\\Phi\end{smallmatrix}$ represents basis columns 1, $1 + 8 = 9$, $1 + 4 = 5$, $1 + 2 = 3$, $1 + 8 + 4 = 13$, $1 + 8 + 2 = 11$, $1 + 4 + 2 = 7$, and $1 + 8 + 4 + 2 = 15$.

Finding the Essential Prime Implicants Directly. In essence, in the previous simplification method we tried all combinations to obtain columns with the most Φ's by a uniform "building-up" process. These columns then represented the prime implicants, and we had to make a prime-implicant table in order to choose the essential prime implicants. Now, however, we shall approach the problem from a different point of view. *We focus our attention on a single column at a time* (a column corresponding to a unit of the given designation number); *if we can show that this column is in a* single *prime implicant, then that prime implicant must be essential* (in such a case we say that the essential prime implicant is based on this column). If we make this examination for each (unit) column, then clearly we shall have determined all *essential* prime implicants, because an essential prime implicant is a prime implicant that has at least one (unit) column contained in no other prime implicant.

To find whether a column is in a single prime implicant, first find all columns that differ from it in only one power position, i.e., by only 1 bit. Suppose that there are p such power positions (that is, p columns). Then determine whether or not columns occur that have *all* possible combinations of 0 and 1 in these power positions, that is, 2^p columns altogether. If so, the original column is in an essential prime implicant.

Suppose, for example, we are given a column $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$. The first step is to find all columns which differ from this by only 1 bit. Suppose that

columns $\begin{smallmatrix}1\\0\\0\\1\end{smallmatrix}\ \begin{smallmatrix}1\\0\\1\\0\end{smallmatrix}$, and $\begin{smallmatrix}1\\1\\0\\0\end{smallmatrix}$ also occurred (but no others that differed from $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ by 1 bit). Then—and this is the crux of our whole argument—the column $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ will be in a single prime implicant (i.e., essential) only if columns $\begin{smallmatrix}1\\0\\1\\1\end{smallmatrix}\ \begin{smallmatrix}1\\1\\0\\1\end{smallmatrix}\ \begin{smallmatrix}1\\1\\1\\0\end{smallmatrix}$, and $\begin{smallmatrix}1\\1\\1\\1\end{smallmatrix}$ also occurred, for then we would have the total combination written as $\begin{smallmatrix}1\\\phi\\\phi\\\phi\end{smallmatrix}$. The essential prime implicant in this case would be simply A. On the other hand, suppose, say, that $\begin{smallmatrix}1\\1\\1\\1\end{smallmatrix}$ were missing; then we have the following reduction by our above simplification method: $\begin{smallmatrix}1&1&1&1\\0&0&1&1\\0&1&0&1\\0&1&1&0\end{smallmatrix}$ then $\begin{smallmatrix}1&1&1&\phi\\0&0&\phi&1\\0&\phi&0&1\\\phi&0&0&1\end{smallmatrix}$, then $\begin{smallmatrix}1&1&1\\0&\phi&\phi\\\phi&0&\phi\\\phi&\phi&0\end{smallmatrix}$; that is, $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ is in *three* prime implicants.

Systematic Method of Procedure. A systematic method of procedure is as follows: Write out the columns corresponding to units of the designation number. For any given column under consideration find another column that also occurs differing in 1 bit, say in power position p_1. Next find a second column that occurs differing from the original columns in 1 bit, say in power position p_2; then make sure the other column occurs that completes the 2^2 combinations of 0 and 1 in both these power positions, p_1 and p_2. Next find a third column that occurs differing in 1 bit from the original column, say in power position p_3; make sure all other columns also occur that complete all 2^3 combinations of 0 and 1 in these three power positions p_1, p_2, p_3; and so forth. This process will stop either when for a certain number of power positions all combinations do not occur (in which case our original column is in more than one prime implicant) or when no further columns occur that differ from the original column in only one power position (in which case the original column is in an essential prime implicant).

This process can be tabulated. Let us illustrate with the following designation number:

1	3	4	5	6	7	9	11	13	14	15
0	1	1	1	1	1	0	1	0	1	1

The first step is to make the prime-implicant table format with the successive power positions to the right:

1 4	3 5 6 9	7 11 13 14	15	Power positions D 8	C 4	B 2	A 1

We have grouped the column numbers according to the number of units in the column. Start, say, with the leftmost column as the original given column, and write its binary form (as a row for convenience) under the power positions; place an x under the 1 to record that this is the column under consideration:

1 4	3 5 6 9	7 11 13 14	15	D 8	C 4	B 2	A 1
x				**0**	**0**	**0**	**1**

Now ask: Does the column occur that differs from the original column in power position 8, that is, does column $1 + 8 = 9$ occur? It does; so we record this by placing an 8 under column number 9 and changing 0001 to $\phi 001$ since both these columns do occur:

1 4	3 5 6 9	7 11 13 14	15	D 8	C 4	B 2	A 1
x	**8**			**ϕ**	0	0	1

Next ask: Does the column occur that differs from the original column in power position 4, that is, does column $1 + 4 = 5$ occur? Yes. Do all 2^2 combinations occur? To answer this, note that $9 + 4 = 13$, and column 13 does occur, so the answer is "yes." Thus we can change $\phi 001$ to $\phi\phi 01$, and record the work as follows:

1 4	3 5 6 9	7 11 13 14	15	D 8	C 4	B 2	A 1
x	**4** 8	**4**		ϕ	**ϕ**	0	1

Next, try power position 2: column $1 + 2 = 3$ occurs; so try $5 + 2 = 7$, $9 + 2 = 11$, and $13 + 2 = 15$. All these columns occur, and hence all combinations of 0 and 1 occur in the three power positions 2, 4, and 8; hence we have

1 4	3 5 6 9	7 11 13 14	15	D 8	C 4	B 2	A 1
x	**2** 4 8	**2** **2** 4	**2**	ϕ	ϕ	**ϕ**	1

Next try power position 1. Note that, since column 1 has a unit in power position 1, the column that differs from column 1 in this position only is column $1 - 1 = 0$. But column 0 does not occur, and therefore column 1 is in a single, hence essential, prime implicant, namely, A.

At this point we are ready to see the *main advantage of the present method.* Since we have found an essential prime implicant that covers

columns 1, 3, 5, 9, 7, 11, 13, and 15, *we need no longer worry about covering them.* So our problem is reduced to covering only columns 4, 6, and 14. Thus we *check off* all columns except these and choose column 4 as our next given column,

✓		✓	✓		✓	✓	✓	✓		✓	*D*	*C*	*B*	*A*	Essential
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1	prime implicant
x		2	4		8	2	2	4		2	Φ	Φ	Φ	1	**A**
	x										**0**	**1**	**0**	**0**	

The process for column 4 is the same. *However, note that, in considering a power position corresponding to a zero in the original column, we* add *this power of* 2 *in order to determine what combinations of columns should occur, i.e., change the zero to a unit; if the power position corresponds to a unit in the original column, we would* subtract *this power of* 2 *in order to determine what columns should occur, i.e., change the* 1 *to a* 0. The work for column 4 becomes

✓	✓	✓	✓	✓	✓	✓		✓	*D*	*C*	*B*	*A*	Essential
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1	prime implicant
x		2	4		8	2	2	4		2	Φ	Φ	Φ	1	*A*
	x		**1**	**2**		**1**					0	1	Φ	Φ	$\bar{D} \cdot C$
											Φ	Φ	Φ	Φ	**C · B**

Here column $4 + 8 = 12$ did not occur, nor did column $4 - 4 = 0$. But column $4 + 2 = 6$ did occur, and so did columns $4 + 1 = 5$ and $6 + 1 = 7$. Thus we can check columns 4 and 6. Only column 14 remains unchecked; so we choose this column as the next given column. The work appears as follows:

✓	✓	✓	✓	✓	✓	✓	✓	✓	*D*	*C*	*B*	*A*	Essential
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1	prime implicant
x		2	4		8	2	2	4		2	Φ	Φ	Φ	1	*A*
	x		1	2		1					0	1	Φ	Φ	$\bar{D} \cdot C$
				8		**1**			x	**1**	Φ	**1**	**1**	Φ	**C · B**

Column $14 - 8 = 6$ occurs; column $14 - 4 = 10$ does not occur; column $14 - 2 = 12$ does not occur; columns $14 + 1 = 15$, $6 + 1 = 7$ do occur. Hence $C \cdot B$ is an essential prime implicant.

Examples and Discussion. Consider the following designation number, and let us find its simplest sum-of-products form:

	1		3			5	6	7		8																	
0	1	0	1		0	1	1	1		1	0	0	0		0	0	0	0									
									16	17		19		20	21	22	23					26	27			30	
									1	1	0	1		1	1	1	1			0	0	1	1			0	0	1	0

The work format appears as follows:

✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		*E*	*D*	*C*	*B*	*A*	Essential
1	8	16	3	5	6	17	20	7	19	21	22	26		23	27	30	16	8	4	2	1	prime implicants
x			2	4		16		2	2	4				2			Φ	0	Φ	Φ	1	$\bar{D} \cdot A$
	x																0	1	0	0	0	$\bar{E} \cdot D \cdot \bar{C} \cdot \bar{B} \cdot \bar{A}$
		x				1	4			1							1	0	Φ	0	Φ	$E \cdot \bar{D} \cdot \bar{B}$
				x				1			16			1			Φ	0	1	1	Φ	$\bar{D} \cdot C \cdot B$
												x			①	4	1	1	Φ	1	0	
									⑧			1			x		1	1	0	1	Φ	
											8	④				x	1	Φ	1	1	0	

Note that in adding (or subtracting) to a column number the result is always in an adjacent group of column numbers (to the right in adding, to the left in subtracting). For column 1 we find $1 + 16 = 17$, which occurs; $1 + 8 = 9$, which does not occur; $1 + 4 = 5$, $17 + 4 = 21$, both of which occur; $1 + 2 = 3$, $5 + 2 = 7$, $17 + 2 = 19$, and $21 + 2 = 23$, all of which occur; but $1 - 1 = 0$ does not occur. Thus an essential prime implicant has been found, namely, $\bar{D} \cdot A$, and column numbers 1, 3, 5, 17, 7, 19, 21, and 23 need not be used as a given column again. Note that nothing combined with column 8. When column number 26 is the given column, note that neither $26 - 16 = 10$ nor $26 - 8 = 18$ occurs; but $26 + 4 = 30$ occurs. Continuing, $26 - 2 = 24$ does not occur, but $26 + 1 = 27$ does occur; however, $30 + 1 = 31$ does not occur, and *no* essential prime implicant is found—we encircle the 1 to show that it did not combine with 30. Similarly, when 27 is the given column number, $27 - 1 = 26$ occurs, and $27 - 8 = 19$ occurs; but $26 - 8 = 18$ does not occur, and so we have encircled the 8. Three columns are not covered by essential prime implicants, 26, 27, and 30; these can be covered by combinations of the last three implicants (or those indicated by the circled numbers). Hence one result is

$$A \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot D \cdot \bar{E} + \bar{B} \cdot \bar{D} \cdot E + B \cdot C \cdot \bar{D} + B \cdot \bar{C} \cdot D \cdot E + \bar{A} \cdot B \cdot D \cdot E$$

Consider a six-variable problem with units in the following designation-number positions: 1, 3, 4, 5, 7, 13, 17, 19, 20, 23, 31, 33, 35, 36, 37, 39, 44, 45, 49, 51, 52, 53, 55, 60, and 61. The work is shown in Table 10-4, in which the columns are grouped appropriately. Here we have introduced a variation of the present method. For example, starting with column number 1 we find that $1 + 32 = 33$ occurs; $1 + 16 = 17$ and $33 + 16 = 49$ both occur; $1 + 8 = 9$ does not occur; but although $1 + 4 = 5$ occurs, $17 + 4 = 21$ does not occur, and so we encircle 4. Now instead of proceeding as above let us see if we can at this point make further use of the information so far obtained, namely, Φ Φ 0 0 0 1.

TABLE 10-4. THE SIMPLEST SUM OF PRODUCTS IS $A \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot \bar{B} \cdot C \cdot \bar{E} + \bar{B} \cdot C \cdot F + A \cdot B \cdot C \cdot E \cdot \bar{F} + \bar{D} \cdot B \cdot A$, WHERE ONLY THE LAST TERM IS NOT AN ESSENTIAL PRIME IMPLICANT†

✓ 1	✓ 4	✓ 3	✓ 5	✓ 17	✓ 20	✓ 33	✓ 36	✓ 7	✓ 13	✓ 19	✓ 35	✓ 37	✓ 44	✓ 49	✓ 52	✓ 45	✓ 51	✓ 53	✓ 60	✓ 31	✓ 55	✓ 61	F 32	E 16	D 8	C 4	B 2	A 1	Essential prime implicants
x		2	④	~~16~~		32				2	2			16									Φ	Φ	0	0	Φ	1	$\bar{D} \cdot \bar{C} \cdot A$
	x		①		~~16~~		32								16			2					Φ	Φ	0	1	0	0	$\bar{D} \cdot C \cdot \bar{B} \cdot \bar{A}$
④		x							8			32					8						Φ	0	Φ	1	0	1	$\bar{E} \cdot C \cdot \bar{B} \cdot A$
②		4	②	②		②		x		4	4					8				~~16~~	32		Φ	Φ	0	Φ	1	1	
							8					1	x	8			1			1			1	Φ	Φ	1	0	Φ	$F \cdot C \cdot \bar{B}$
														8						x		1	0	1	Φ	1	1	$\bar{F} \cdot E \cdot C \cdot B \cdot A$

† Note that Φ Φ 0 Φ 1 1, corresponding to $\bar{D} \cdot B \cdot A$, in the fourth row covers the unchecked columns. By our method of procedure this is as good as any other product which covers them, and hence we include $\bar{D} \cdot B \cdot A$ in our simplest sum of products.

To do this, suppose that we transfer to column number 17 as our given column; we indicate this transfer by putting a slash through the 16 below column number 17, as is shown. Then Φ Φ 0 0 0 1 still holds for column 17, and we need not try power 32 or power 16 again or, for that matter, power 4 (why?). But we must try power 8 again, for we have never tried it on this column (why not?). Thus 17 + 8 = 25 does not occur; but 17 + 2 = 19, 1 + 2 = 3, 33 + 2 = 35, 49 + 2 = 51 all occur, giving us Φ Φ 0 0 Φ 1. Finally 17 − 1 = 16 does not occur; so we have discovered an essential prime implicant, *based on column* 17 rather than on the column we started with, column 1. The next two rows of work in Table 10-4 also indicate a transfer to some other column number in finding the essential prime implicant.

Next observe the fourth row of work. Here we meet several further circumstances, which we shall explain by means of the illustration. The original column number is 7; 7 + 32 = 39 occurs; 7 + 16 = 23 and 39 + 16 = 55 occur; 7 + 8 = 15 does not occur; 7 − 4 = 3, 23 − 4 = 19, 39 − 4 = 35, and 55 − 4 = 51 all occur, giving so far Φ Φ 0 Φ 1 1. Next observe that 7 − 2 = 5, 3 − 2 = 1, 19 − 2 = 17, 35 − 2 = 33 all occur but that 23 − 2 = 21 does not. Transferring to 23 as above, we need not try powers 32, 16, 4, or 2 again. Trying 8, we find that 23 + 8 = 31 occurs but that 3 + 8 = 11 does not occur. We could try to transfer once again, completing the crossing off of the 16 below column number 23 for the record. *But we must transfer to a column with the following properties:* (1) *it is not covered by a previously determined essential prime implicant and* (2) *it does not combine with powers that have already been tried and circled.* If the transfer were made to a column already covered, then this column obviously could not be the basis for another essential prime implicant; if the column to which we transferred combined with a circled power, then this power clearly would not again allow the completion of an essential prime implicant. For our illustration it is seen that no further column satisfying Φ Φ 0 Φ 1 1 as well as our conditions 1 and 2 exists. Thus we must go on to a new row of work, i.e., another uncovered column. Note that since we already have determined that no column satisfying Φ Φ 0 Φ 1 1 can be the basis of an essential prime implicant we should not go on to any of these columns either. For our illustration the next given column will be 44, and the rest of the work is completed as above.

10-9. Constraints

The Problem. Up to now we have been considering systems of *independent elementary elements.* That is, for elementary elements with the basis given above, the truth value of each elementary element is independent of the truth value of any other elementary element. It often

happens, however, that this is not the case; the truth value of an elementary element *may* depend on the truth values of the other elementary elements *for a particular problem.* Such a dependence is given in terms of a combined element which is to remain true *for this problem.* For example, in a problem we might always have that $A \rightarrow B$. The effect of such a constraint is to limit the possible combinations of truth values that A and B can take simultaneously. If $A \rightarrow B$ is to hold, then B cannot be false when A is true; i.e., the combination of truth values A true, B false cannot occur.

Summarizing, *a constraint is a logical relation between the elementary elements (i.e., a combined element) which is to be considered as tautologically true (i.e., always true) because of the stated or intrinsic circumstances of the problem.* Put another way, constraints are factually true combined elements which are to remain true throughout a problem and therefore, with respect to this problem, can be considered as tautologies. In this section will be given a systematic computational method that automatically guarantees the maintenance of any desired constraints throughout the solution of a problem, so that all advantages produced by the constraints can be obtained in a straightforward manner.

The effect of constraints is to limit the number of input states; i.e., some simultaneous combinations of truth values are forbidden by the constraints. For example, suppose that, for three variables A, B, and C, no two could both be true simultaneously, and it never occurs that all three are false simultaneously. These constraints can be written as follows: $A \cdot B = 0$, $A \cdot C = 0$, $B \cdot C = 0$, and $\bar{A} \cdot \bar{B} \cdot \bar{C} = 0$.

We have remarked that for n inputs there are 2^n columns in the basis, one column for each of the 2^n possible input states. With constraints, however, some of these 2^n states are forbidden. Hence it seems plausible that in order to take account of the constraints during a problem we should simply cross off those columns in the basis which represent forbidden states of the particular problem—forming a new, reduced, constrained basis. Then we should proceed as described in the previous sections, except that *all computational methods should be performed with respect to this constrained basis.* This is the method for taking account of constraints.

The first problem that arises is: Given a set of constraints as Boolean functions, which columns of the basis are to be crossed off to form the constrained basis? Conversely, if we are given the columns to be crossed off or the constrained basis, what is the Boolean form of the constraints?

Determining the Constrained Basis. Let us consider the first of these problems, to find the constrained basis, given the constraints. Suppose that $f_1(A,B,C, \ldots)$, $f_2(A,B,C, \ldots)$, $\ldots$, $f_n(A,B,C, \ldots)$ are constraints which are all to be tautologically true simultaneously. We form

the logical product of the designation numbers of the constraints,

$$\prod_{j=1}^{n} \#f_j(A,B,C, \ldots) = f_1(A,B,C, \ldots) \cdot f_2 \cdot f_3 \cdot \cdots \cdot f_n$$

It can be seen that the positions of the zeros in the resulting number correspond to input states that can never occur if the constraints are all to remain true. Hence *the constrained basis is formed by crossing off those columns of the usual basis that correspond to the positions of the zeros of the product number.*

For example, consider the designation numbers of the constraints of the above example,

$$\begin{aligned} \#(A \cdot B = 0) &= 1110 \quad 1110 \\ \#(A \cdot C = 0) &= 1111 \quad 1010 \\ \#(B \cdot C = 0) &= 1111 \quad 1100 \\ \#(\bar{A} \cdot \bar{B} \cdot \bar{C} = 0) &= 0111 \quad 1111 \\ \text{Product} \quad & \underline{0110 \quad 1000} \\ & \quad 0 \;\; 3 \quad 5\,6\,7 \end{aligned}$$

Crossing off columns 0, 3, 5, 6, and 7 of the usual basis, we find for the constrained basis

$$\begin{aligned} \#A &= 100 \\ \#B &= 010 \\ \#C &= 001 \end{aligned}$$

Conversely, given a constrained basis, the explicit form of the constraints can be determined as a Boolean function. We shall give this procedure by means of an example. Suppose that the constrained basis is

$$\begin{aligned} \#A &= 0101 \quad 01 \\ \#B &= 0011 \quad 00 \\ \#C &= 0000 \quad 11 \end{aligned}$$

The columns of this constrained basis correspond to the columns in positions 0, 1, 2, 3, 4, and 5 of the standard basis. Hence we know that the designation number of the constraint must have units in these positions,

$$\quad\quad\quad

function with respect to the constrained basis and conversely (2) to find the Boolean function corresponding to a given designation number (in any of the various forms considered above) with respect to the constrained basis. These computations are identical with those discussed above except that all procedures are *with respect to the constrained basis.* However, for clarity we shall illustrate these procedures.

For example, consider the following constrained basis:

$$\begin{aligned} \#A &= 0101 \quad 01 \\ \#B &= 0011 \quad 00 \\ \#C &= 0000 \quad 11 \end{aligned}$$

To find $\#(\bar{A} + B \cdot C)$ with respect to this basis, we have

$$\begin{aligned} \#B &= 0011 \quad 00 \\ \#C &= 0000 \quad 11 \\ \hline \#(B \cdot C) &= 0000 \quad 00 \\ \#\bar{A} &= 1010 \quad 10 \\ \hline \#(\bar{A} + B \cdot C) &= 1010 \quad 10 \end{aligned}$$

Note that in this case it is simply $\#\bar{A}$.

For the converse problem of finding the Boolean function of a given designation number with respect to a constrained basis, the designation number can be given in two different ways: either (1) with the same number of positions as columns of the constrained basis, in which case it is assumed that the positions of the designation number correspond to the columns of the constrained basis, or (2) with 2^n positions, for n elementary elements. In this second case the positions of the given designation numbers that correspond to the missing columns of the constrained basis are crossed off, and the second case is thereby reduced to the first case. For instance, suppose that it were desired to find the Boolean function corresponding to 1010 1011 with respect to the constrained basis given in the above example. Columns 6 and 7 of the standard basis are missing in this constrained basis; hence we cross off positions 6 and 7 of our number: 1010 10 ~~11~~.

The methods given for finding the simplest sum-of-products form can easily be adapted to take constraints into consideration. Every constrained basis column (i.e., eliminated column) can be used freely if it combines with nonconstrained columns corresponding to the units of the designation number. However, positions corresponding to constrained columns do not necessarily have to be covered by a prime implicant (as do the unit positions of the constrained designation number). In other words, the constrained positions can be thought of as units or zeros, whichever helps best to simplify the function.

For example, consider 1111 01 with respect to the constrained basis of the above example. By our first simplification method we have

Constrained columns displayed for possible use

✓	✓✓	✓✓			
0	10	11		0	1
0	01	10		1	1
0	00	01		1	1

✓✓	✓✓✓		
Φ0	11Φ0	11	Formed by combination with constrained columns
0Φ	Φ011	1Φ	
00	0Φ0Φ	Φ1	

Φ	1Φ
Φ	Φ1
0	ΦΦ

The table of prime implicants (omitting columns 6 and 7) becomes

	0	1	2	3	5
$\bar{C}$	0	1	2	3	
A		1		3	5
B			2	3	

whence $\bar{C}$ and A are essential prime implicants whose sum covers the unit positions of the constrained designation number; thus

$$1111 \quad 01 = \#(A + \bar{C})$$

The second method given above for finding the simplest sum of products can likewise be adapted to handle constrained designation numbers. Consider again the number 1111 01 as an example:

0	1	2	3	5	6	7	C 4	B 2	A 1	
x	1	2	1				0	Φ	Φ	$\bar{C}$
	4		2	x		2	Φ	Φ	1	A

Constrained positions

Note that we had to make use of constrained column 7 in order to form all combinations with B (power 2) in the second line of the illustration.

The methods for going from the constrained designation number to other Boolean function forms, e.g., the first and second canonical forms, follow easily.

Constrained Boolean Algebra. An important point arises from the last example. Note that, with respect to the constrained basis given above,

many strange relations hold; for example,

$$A = A \cdot (\bar{B} + \bar{C}) \qquad A + B = A \cdot \bar{B} + B \cdot \bar{C} + A \cdot C$$

Such relations obviously do not hold with respect to the standard unconstrained basis. In other words, with respect to a constrained basis, we obtain a *constrained Boolean algebra* which is different from the unconstrained Boolean algebra described above. That is, additional relations hold in a constrained Boolean algebra that do not hold in the unconstrained algebra—but all relations that hold in the unconstrained algebra also hold in the constrained algebra. In general these additional relations in the constrained algebra often allow Boolean functions to be further simplified; that is, $A \cdot (\bar{B} + \bar{C})$ becomes just A, and $A \cdot \bar{B} + B \cdot \bar{C} + A \cdot C$ becomes $A + B$ with respect to the constrained basis of the above example. However, it is not always possible to simplify a function further; for instance, $\bar{A} \cdot \bar{C} + A \cdot C$ cannot be put into simpler form with respect to the constrained basis of the above example.

10-10. Logical Dependence and Independence

Logical Independence. Boolean functions are logically independent (with respect to each other) if they can take on all possible combinations of truth values. There are 2^n possible combinations of truth values that n Boolean functions can have (since each can take either of two truth values). Hence, in order to test the independence of n Boolean functions, we need examine only the combinations of truth values that can occur: if all 2^n are possible, then they are independent; otherwise they are not. This can be done by means of the designation numbers for the Boolean functions. The procedure is illustrated by means of an example.

Suppose that we desire to know whether or not the two functions $A \cdot B + \bar{A} \cdot \bar{B}$ and $\bar{B}$ are independent. Write their designation numbers on successive rows,

$$\begin{aligned} \#(A \cdot B + \bar{A} \cdot \bar{B}) &= 1001 \\ \#\bar{B} &= 1100 \end{aligned}$$

The columns represent all possible combinations of truth values. Since all four column combinations are there, we can say that $A \cdot B + \bar{A} \cdot \bar{B}$ and $\bar{B}$ are independent.

As another example, consider $A \cdot B + \bar{B} \cdot \bar{C}$ and $\bar{A} \cdot \bar{B} + B \cdot C$. We write

$$\begin{aligned} \#(A \cdot B + \bar{B} \cdot \bar{C}) &= 1101 \quad 0001 \\ \#(\bar{A} \cdot \bar{B} + B \cdot C) &= 1000 \quad 1011 \end{aligned}$$

Here $\begin{matrix} 1 & 1 & 0 \\ 1' & 0' & 1' \end{matrix}$ and $\begin{matrix} 0 \\ 0 \end{matrix}$ all appear; so the functions are, indeed, independent.

Consider next the three functions $A \cdot B + \bar{A} \cdot \bar{B}$, $\bar{B}$, and $A \cdot \bar{B} + \bar{A} \cdot B$. Write

$$\begin{aligned} \#(A \cdot B + \bar{A} \cdot \bar{B}) &= 1001 \\ \#\bar{B} &= 1100 \\ \#(A \cdot \bar{B} + \bar{A} \cdot B) &= 0110 \end{aligned}$$

Here we need $2^3 = 8$ possible combinations in order that the *three* functions be independent. But only four columns appear; hence the functions are *not* independent.

Logical Dependence. Boolean functions that are not logically independent are said to be *logically dependent.* This means that there must be some logical relations between them that can be expressed as a combined element. For instance, we know that $A \cdot B$ and $A + B$ are logically dependent because clearly $A \cdot B \rightarrow (A + B)$. Given a set of n Boolean functions that are logically dependent, the explicit form of this dependence can be found by means of the designation numbers, in two steps. This first step is to write their designation numbers in successive rows. In this array of designation numbers cross off columns so that no two identical columns remain. Since the functions are dependent, there will be fewer than 2^n columns remaining. Put these columns in their standard order, and consider this resulting array as a *constrained basis.* The second step is to find the constraint in the same manner as in the previous section, remembering that the rows are labeled by the given functions (and not elementary elements). For example, consider the functions $\bar{B} \cdot \bar{C} + \bar{A} \cdot C$, $\bar{A} \cdot \bar{C} + \bar{B} \cdot C$, and $\bar{B}$. Let us denote these by f_1, f_2, and f_3, respectively. Then we have

$$\begin{aligned} \#f_1 &= 1100 \quad 01 \\ \#f_2 &= 1010 \quad 10 \\ \#f_3 &= 1100 \quad 10 \end{aligned}$$

and on rearrangement in standard order we obtain the constrained basis:

$$\begin{array}{rll} & 012 & 567 \\ \#f_1 = & 010 & 101 \\ \#f_2 = & 001 & 011 \\ \#f_3 = & 000 & 111 \end{array}$$

Hence the designation number of the constraint is 1110 0111. This designation number is to be evaluated with respect to the following standard (unconstrained) basis:

$$\begin{aligned} \#f_1 &= 0101 \quad 0101 \\ \#f_2 &= 0011 \quad 0011 \\ \#f_3 &= 0000 \quad 1111 \end{aligned}$$

whence we have $1110\ 0111 = \#[(\bar{f}_1 + \bar{f}_2) \cdot \bar{f}_3 + (f_1 + f_2) \cdot f_3]$. Hence the logical dependence between f_1, f_2, and f_3 is given by

$$\#[(\bar{f}_1 + \bar{f}_2) \cdot \bar{f}_3 + (f_1 + f_2) \cdot f_3] = \#I$$

as can be easily verified by substituting for f_1, f_2, and f_3 their corresponding functions in terms of A, B, and C.

10-11. Boolean Matrices

Preliminaries: Boolean Matrix Operations. For our purposes, by a *Boolean matrix* A is meant a rectangular array of elements a_{ij} each of which can have only the value 0 or 1. *Boolean matrix* multiplication, denoted by $\otimes$, is defined in a manner directly analogous to ordinary matrix multiplication, except that *logical addition* and *logical multiplication* replace ordinary addition and multiplication. Thus we write

$$A \otimes B = C \qquad \text{where } c_{ij} = \sum_k a_{ik} \cdot b_{kj} \tag{10-1}$$

For example,

$$\begin{pmatrix}01\\10\\00\end{pmatrix} \otimes \begin{pmatrix}101\\010\end{pmatrix} = \begin{pmatrix}010\\101\\000\end{pmatrix}$$

We can define the *complement* $\bar{A}$ of a Boolean matrix A by

$$\bar{A} = C \qquad \text{where } c_{ij} = \bar{a}_{ij} \tag{10-2}$$

For example, if

$$A = \begin{pmatrix}101\\010\end{pmatrix} \qquad \text{then } C = \bar{A} = \begin{pmatrix}010\\101\end{pmatrix}$$

If $+$ denotes logical addition and $\cdot$ logical multiplication, then we can define the *logical sum* and *logical product* of two Boolean matrices by

$$A + B = C \qquad \text{where } c_{ij} = a_{ij} + b_{ij} \tag{10-3}$$
$$A \cdot B = C \qquad \text{where } c_{ij} = a_{ij} \cdot b_{ij} \tag{10-4}$$

For example,

$$\begin{pmatrix}101\\010\end{pmatrix} + \begin{pmatrix}110\\001\end{pmatrix} = \begin{pmatrix}111\\011\end{pmatrix}$$
$$\begin{pmatrix}101\\010\end{pmatrix} \cdot \begin{pmatrix}110\\001\end{pmatrix} = \begin{pmatrix}100\\000\end{pmatrix}$$

The transpose A^t of A is found by interchanging the rows and columns of A; that is, if

$$A^t = C \qquad \text{then } c_{ij} = a_{ji} \tag{10-5}$$

The relation *implies* is said to hold between two Boolean matrices A and B, denoted by $A \rightarrow B$, if there are no i and j such that $b_{ij} = 0$ and $a_{ij} = 1$. For example,

$$\begin{pmatrix}101\\010\end{pmatrix} \rightarrow \begin{pmatrix}111\\110\end{pmatrix}$$

If both $A \rightarrow B$ and $B \rightarrow A$, that is, if the elements of A and of B are identical, then we say that A *equals* B, denoted by $A = B$. For some useful properties of these operations and relations see Exercises 10-98 to 10-105.

Solutions to Boolean Matrix Equations. The following theorem is of importance to the development of our transformation theory (see below):

THEOREM 1. If the Boolean matrix equation

$$A \otimes X = B \tag{10-6}$$

has a solution X, then

$$X' = \overline{A^t \otimes \bar{B}} \tag{10-7}$$

is a solution, and in fact if X is any other solution, then

$$X \rightarrow X' \tag{10-8}$$

Furthermore, if X' is not a solution, then there are no solutions and

$$A \otimes X' \rightarrow B \tag{10-9}$$

Proof.† Equation (10-6) is equivalent to the set of equations

$$\sum_j a_{ij} \cdot x_{jk} = b_{ik} \tag{10-10}$$

If a solution exists, consider Eq. (10-10) for a fixed value of k and those i for which $b_{ik} = 0$: then for each j such that $a_{ij} = 1$, the corresponding x_{jk} must be 0; for each j such that $a_{ij} = 0$, the x_{jk} can be 0 or 1. For those i for which $b_{ik} = 1$, there must be at least one value for j for which both $a_{ij} = 1$ and $x_{jk} = 1$. By the assumption that a solution exists, it will be obtained if we set $x_{jk} = 0$ only when necessary, i.e., when $b_{ik} + \bar{a}_{ij} = 0$ for at least one i, that is, when $\prod_i (b_{ik} + \bar{a}_{ij}) = 0$; otherwise $x_{jk} = 1$. Denoting this set of x_{jk} by the x'_{jk}, we have

$$x'_{jk} = \prod_i (b_{ik} + \bar{a}_{ij}) = \overline{\sum_i \bar{b}_{ik} \cdot a_{ij}}$$

This of course holds for each value of k, whence Eq. (10-7) follows. Clearly also X' has units in at least those positions where any other solution X has units, and hence Eq. (10-8) follows. If no solution exists, then clearly, by the construction of X', Eq. (10-9) follows, which completes the proof.

Observe that if solutions exist to Eq. (10-6), then by Eq. (10-8) the matrix X' may be considered as the (unique) least upper bound of the

† See, for example, R. D. Luce, Note on Boolean Matrix Theory, *Proc. Am. Math. Soc.*, vol. 3, pp. 382–388, 1952.

set of all solutions. In general there is no greatest lower bound to the set of all solutions. For example, it can be shown† that the equation

$$\begin{pmatrix}101\\011\end{pmatrix} \otimes X = \begin{pmatrix}10\\11\end{pmatrix} \tag{10-11}$$

has five solutions as diagramed in Fig. 10-11, where

$$X' = \begin{pmatrix}10\\11\\10\end{pmatrix}$$

COROLLARY 1*a*. If the Boolean matrix equation $Y \otimes C = D$ has a solution Y, then $Y' = \overline{\bar{D} \otimes C^t}$ is a solution, and in fact if Y is any other

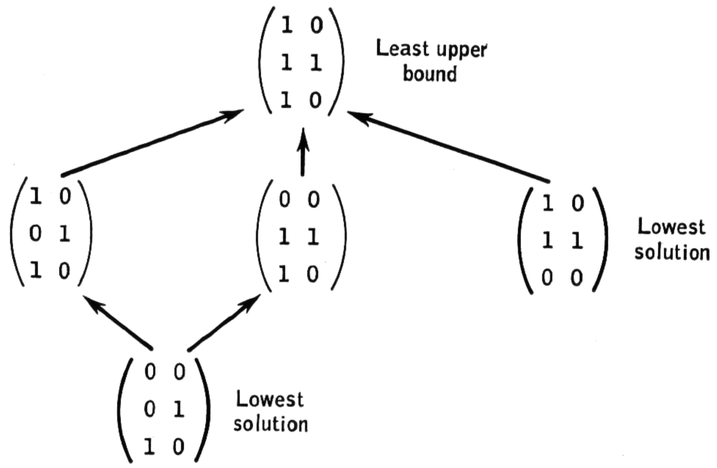


FIG. 10-11. The five solutions to Eq. (10-11); the arrows indicate implication relations.

solution, then $Y \rightarrow Y'$. Furthermore, if Y' is not a solution, then there are no other solutions and $Y' \otimes C \rightarrow D$. The proof follows from the theorem by the relation $(Y \otimes C)^t = C^t \otimes Y^t$.

The Unitary Matrix Equation. As will be shown below, the Boolean transformation matrix has exactly one unit in each column; such a matrix U is called a *unitary matrix*. A unitary matrix will perform a kind of permutation of the columns of a matrix M multiplied on the right. That is, if $M \otimes U = B$, and if $u_{ji} = 1$, then column j of M will appear as column i of B. For example,

$$\begin{pmatrix}1000\\1100\\1110\\1111\end{pmatrix} \otimes \begin{pmatrix}0110\\0000\\0001\\1000\end{pmatrix} = \begin{pmatrix}0110\\0110\\0111\\1111\end{pmatrix}$$

† A method for systematically generating all the solutions was developed by W. R. Smith; see R. S. Ledley, "Digital Computer and Control Engineering," pp. 445–447, McGraw-Hill Book Company, Inc., New York, 1960; see also Exercise 10-116 below.

Since clearly the results of complementing a matrix and then permuting its columns and the results of permuting its columns and then complementing it are the same, then

$$\overline{M \otimes U} = \bar{M} \otimes U \tag{10-12}$$

Thus the unitary matrix equation

$$M \otimes U = B \tag{10-13}$$

can be equivalently written as

$$\bar{M} \otimes U = \bar{B} \tag{10-14}$$

Solutions to the Unitary Matrix Equation. The characteristics of the set of solutions to Eq. (10-13), considering M as the unknown, are of importance for our purposes.

THEOREM 2. The set of solutions of Eq. (10-13), considering M as the unknown, has a unique least upper bound given by

$$M' = \overline{\bar{B} \otimes U^t} \tag{10-15}$$

and a unique greatest lower bound given by

$$M^* = B \otimes U^t \tag{10-16}$$

when solutions exist at all. Every other solution M can be found from the relation

$$M^* \rightarrow M \rightarrow M' \tag{10-17}$$

Proof. Equations (10-15) and (10-16) follow from Corollary 1*a* applied to Eqs. (10-13) and (10-14). Next note that Eq. (10-16) gives the complement of the least-upper-bound solution to Eq. (10-14). But any solution $\bar{M}$ of Eq. (10-14) is also (upon complementation) a solution of Eq. (10-13), and conversely. Hence M^* given by Eq. (10-16) must be the greatest lower bound of solutions to Eq. (10-13). By the meanings of least upper bound and greatest lower bound, Eq. (10-17) follows, completing the proof. For example, consider the unitary matrix equation

$$M \otimes \begin{pmatrix}0100\\1001\\0000\\0010\end{pmatrix} = \begin{pmatrix}1011\\1001\\1101\end{pmatrix} \tag{10-18}$$

Here $M' = \begin{pmatrix}0111\\0110\\1110\end{pmatrix}$ and $M^* = \begin{pmatrix}0101\\0100\\1100\end{pmatrix}$

and the set of all eight possible solutions is shown in Fig. 10-12.

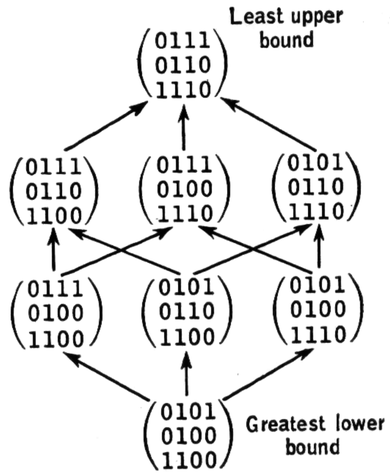


FIG. 10-12. The eight solutions to Eq. (10-18); the arrows indicate implication relations.

It is frequently important to solve a unitary matrix equation for the unitary matrix itself as the unknown. In this connection we have the following theorem:

THEOREM 3. The set of unitary matrix solutions to Eq. (10-13), considering U as the unknown matrix, consists of those *unitary matrices* U such that

$$U \rightarrow R' \cdot R^* \tag{10-19}$$

where $$R' = \overline{M^t \otimes \bar{B}} \quad \text{and} \quad R^* = \overline{\bar{M}^t \otimes B} \tag{10-20}$$

when such solutions exist at all.

Proof.† From Theorem 1 applied to Eq. (10-13) we have $M \otimes R' \rightarrow B$, where R' is given by Eqs. (10-20). If a unitary matrix U is such that

$$U \rightarrow R' \qquad \text{then } M \otimes U \rightarrow B \tag{10-21}$$

Similarly from Theorem 1 and Eq. (10-14) we have $\bar{M} \otimes R^* \rightarrow \bar{B}$, where R^* is given by Eqs. (10-20). If a unitary matrix U is such that

$$U \rightarrow R^* \qquad \text{then } \bar{M} \otimes U \rightarrow \bar{B} \tag{10-22}$$

Recalling that if $P \rightarrow Q$ then $\bar{Q} \rightarrow \bar{P}$, we find from Eq. (10-22), using Eq. (10-12), that

$$B \rightarrow M \otimes U \tag{10-23}$$

Thus if U implies both R' and R^*, that is, if U is such that Eq. (10-19) holds, then from Eqs. (10-21) and (10-23) we have

$$M \otimes U = B$$

which completes the proof.

† See R. S. Ledley, Digital Computational Methods in Symbolic Logic, with Examples in Biochemistry, *Proc. Natl. Acad. Sci.*, vol. 41, no. 7, pp. 498–511, July, 1955.

10-12. Boolean Transformations and the Fundamental Formulas

Boolean Algebraic Transformations. Consider now, for example, a Boolean algebraic transformation from variables f_1, f_2, f_3 to variables A_1, A_2, and A_3, where

$$\begin{aligned} f_1 &= A_2 + \bar{A}_3 \\ f_2 &= (A_1 + A_2) \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + (A_1 + A_2) \cdot A_3 \end{aligned} \tag{10-24}$$

Then any Boolean function $F(f_1,f_2,f_3)$ can be written as a function $E(A_1,A_2,A_3)$ in terms of the new variables A_1, A_2, A_3 by means of the transformation. For example,

$$F = f_1 \cdot f_3 + \bar{f}_2 \tag{10-25}$$

becomes by the above transformation

$$E = A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2 + A_3 \tag{10-26}$$

We desire to find a Boolean transformation matrix R which when multiplied by the designation number F of a Boolean function of variables $f_1, \ldots, f_J$ will result in the designation number E of the transformed function of the new variables $A_1, \ldots, A_I$. That is, we wish to find a matrix R such that†

$$F \otimes R = E \tag{10-27}$$

where F is with respect to $b[f_i, \ldots, f_J]$ and E is with respect to $b[A_1, \ldots, A_I]$. The method for determining R is best described by means of an illustration. Consider the transformation described by Eqs. (10-24), from which the following array is formed:

i	*0123*	*4567*
A_1 =	0101	0101
A_2 =	0011	0011
A_3 =	0000	1111
f_1 =	1111	0011
f_2 =	0111	0000
f_3 =	0001	0111
j	*1337*	*0455*

(10-28)

Here $b[A_1,A_2,A_3]$ appears above the line, and the designation numbers of f_1, f_2, and f_3 with respect to $b[A_1,A_2,A_3]$ appear below the line; the italic numbers on top denote the basis columns of $b[A_1,A_2,A_3]$, while those on the bottom represent basis columns of $b[f_1,f_2,f_3]$. Observe that each jth

† In the remainder of the chapter we drop the sign # and use the letters A, E, F, etc., alone to represent the designation numbers (or matrices) as well as the functions.

column of $b[f_1,f_2,f_3]$ corresponds to some column i of $b[A_1,A_2,A_3]$ just above; therefore, to find E, we must permute the bits of F according to this correspondence. The permutation can be accomplished by a unitary matrix R which has $r_{ji} = 1$ if column j corresponds to column i in the array of Eq. (10-28) or otherwise has $r_{ji} = 0$. Thus for the above example

$$R = \begin{array}{c} i \\ j \end{array} \begin{array}{c} \begin{array}{cc} 0123 & 4567 \end{array} \\ \begin{pmatrix} 0000 & 1000 \\ 1000 & 0000 \\ 0000 & 0000 \\ 0110 & 0000 \\ 0000 & 0100 \\ 0000 & 0011 \\ 0000 & 0000 \\ 0001 & 0000 \end{pmatrix} \end{array} \begin{array}{l} 0 \\ 1 \\ 2 \\ 3 \\ \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \tag{10-29}$$

Hence for F as in Eq. (10-25) we find

$$(1100 \quad 1101) \otimes R = (1001 \quad 1111)$$

where the right-hand side is the designation number of E with respect to $b[A_1,A_2,A_3]$, from which $E = A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2 + A_3$.

Clearly, multiple transformations can be handled in an analogous manner. For example, if we first transform variables $f_1, \ldots, f_J$ into variables $g_1, \ldots, g_K$ by R^f and then transform $g_1, \ldots, g_K$ into $h_1, \ldots, h_L$ by R^g, and finally $h_1, \ldots, h_L$ into $A_1, \ldots, \bar{A}_I$ by R^h, then we can write

$$F \otimes R^f \otimes R^g \otimes R^h = E \tag{10-30}$$

which is a generalization of Eq. (10-27).

Most frequently the transformation does not involve all the variables under consideration. For instance, the variables may be $f_1, \ldots, f_J, X_1, \ldots, X_K$, where only $f_1, \ldots, f_J$ are to be transformed in terms of, say, $A_1, \ldots, A_I$; then the transform of $F(f_1, \ldots, f_J, X_1, \ldots, X_K)$ will be $E(A_1, \ldots, A_I, X_1, \ldots, X_K)$. In this case the designation number of F (and E) can be written in full matrix form† instead of vector form; so the columns of the matrix correspond to values of $f_1, \ldots, f_J$, and the rows to values of $X_1, \ldots, X_K$. Such a matrix, when substituted into Eq. (10-27), will only have its columns permuted by R; that is, only the dependence of F on $f_1, \ldots, f_J$ will be transformed, as desired. For example, let F be the function $f_1 \cdot f_3 \cdot X_1 + \bar{f}_2 \cdot \bar{X}_2$, and find

† See R. S. Ledley, Boolean Matrix Equations in Digital Circuit Design, *IRE Trans. on Electronic Computers*, vol. EC-8, no. 2, pp. 131–139, June, 1959.

j	0123	4567
f_1	0101	0101
f_2	0011	0011
f_3	0000	1111

$$F = \begin{pmatrix} 1100 & 1100 \\ 1100 & 1101 \\ 0000 & 0000 \\ 0000 & 0101 \end{pmatrix} \begin{array}{cc|l} 0 & 0 & 0\,k \\ 1 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 1 & 3 \end{array} \qquad \begin{array}{c} \\ \\ \\ \\ X_1X_2 \end{array} \tag{10-31}$$

where we have separated $b[f_1,f_2,f_3,X_1,X_2]$ into two parts, giving on top the corresponding values of f_1, f_2, f_3, and on the side those of X_1 and X_2.† Then, if f_1, f_2, and f_3 are transformed as in Eqs. (10-24), we find, using F and R as in Eqs. (10-31) and (10-29), respectively, substituting into Eq. (10-27), that

$$F \otimes R = \begin{pmatrix} 1000 & 1111 \\ 1001 & 1111 \\ 0000 & 0000 \\ 0001 & 0011 \end{pmatrix} = E \tag{10-32}$$

which is seen to be $E = (A_1 + A_3) \cdot A_2 \cdot X_1 + (\bar{A}_1 \cdot \bar{A}_2 + A_3) \cdot \bar{X}_2$ in terms of $b[A_1,A_2,A_3,X_1,X_2]$.

The Fundamental Formulas. Summarizing, Eqs. (10-27) and (10-30) give a Boolean matrix representation for Boolean algebraic transformations. Clearly, however, if we are given the transformation and the original function F, the problem of finding the transformed function E is rather trivial and can be solved algebraically by direct substitution. The importance of Eqs. (10-27) and (10-30) becomes clear, however, when the nontrivial problems associated with solving for F or for R are considered. These considerations lead us to the next theorem.

THEOREM 4. If a Boolean function $F(f_1, \ldots, f_J, X_1, \ldots, X_K)$ is related to a Boolean function $E(A_1, \ldots, A_I, X_1, \ldots, X_K)$ by means of the transformation $f_1 = f_1(A_1, \ldots, A_I), \ldots, f_J = f_J(A_1, \ldots, A_I)$, then

$$F \otimes R = E \tag{10-33}$$

where F and E represent the respective designation-number matrices for the functions F and E as described above and R is the unitary permutation or incidence matrix related to the transformation as described above. Furthermore, if R and E are known, then F can be found as satisfying

$$F^* \rightarrow F \rightarrow F' \tag{10-34}$$

where $F' = \overline{\bar{E} \otimes R^t}$ and $F^* = E \otimes R^t$

† Note that this is essentially a Veitch chart for F. See E. W. Veitch, A Chart Method for Simplifying Truth Functions, *Proc. Assoc. Computing Mach.*, May 2–3, 1952, meeting, Pittsburgh, Pa., pp. 127–133.

If no such F exists, then

$$F' \otimes R \rightarrow E \rightarrow F^* \otimes R \tag{10-35}$$

If on the other hand F and E are known, then R can be found from

$$R \rightarrow R' \cdot R^* \tag{10-36}$$

where $R' = \overline{F^t \otimes \bar{E}}$ and $R^* = \overline{\bar{F}^t \otimes E}$

If no such R exists (i.e., if R' or R^* has a column of all zeros), then

$$F \otimes R' \rightarrow E \rightarrow F \otimes R^* \tag{10-37}$$

Proof. The proof of Eq. (10-33) was discussed above; Eqs. (10-34), (10-35), (10-36), and (10-37) follow directly from applying Theorems 2 and 3 to Eq. (10-33).

Equations (10-34) and (10-36) are called the *fundam ntal formulas*, because they are the equations most frequently utilized when applying Boolean matrices to logical circuit design and other problems.

Boolean Algebraic Equations. Consider a Boolean equation of the general form

$$G(A_1, \ldots, A_I, X_1, \ldots, X_N) = H(A_1, \ldots, A_I, X_1, \ldots, X_N) \tag{10-38}$$

where the unknowns $X_1, \ldots, X_N$ are to be found as functions of the $A_1, \ldots, A_I$ so that the equation will hold. From the meaning of the "solution to an equation," we desire to find functions $f_1, \ldots, f_N$ such that, *if*

$$X_1 = f_1, \ldots, X_N = f_N$$

then Eq. (10-38) holds; that is, f functions such that $X_1 = f_1, \ldots, X_N = f_N$ *imply* that Eq. (10-38) holds. Recasting this in terms of our matrix formulation, if we let F represent

$$(X_1 = f_1) \cdot (X_2 = f_2) \cdot \cdots \cdot (X_N = f_N) \tag{10-39}$$

and let E represent Eq. (10-38), then we want F, with the appropriate transformation of each f_j in terms of the A_i, to imply E—that is,

$$F \otimes R \rightarrow E \tag{10-40}$$

Here F and E are known, and the determination of R will give the desired functions f_j in terms of the A_i. Thus the matrix R is given by all R such that $R \rightarrow R'$, where R' is determined from the left-hand form of Eqs. (10-36).

To find the matrix for F (or E) we must first observe that the designation number for the general form $P = Q$ has a unit only in those positions

where both P and Q have units or both P and Q have zeros. Then, for example, with respect to $b[f_1,f_2,X_1,X_2]$ we have

$$(X_1 = f_1) \cdot (X_2 = f_2) = \begin{pmatrix}1010\\0101\\1010\\0101\end{pmatrix} \cdot \begin{pmatrix}1100\\1100\\0011\\0011\end{pmatrix} = \begin{pmatrix}1000\\0100\\0010\\0001\end{pmatrix} \tag{10-41}$$

A further important observation can be made from this example, namely, that the matrix for a function F of the form (10-39) will in general be "diagonal," i.e., will have units only on the diagonal. An immediate consequence is that the solution to Eq. (10-40) as given by the left-hand form of Eqs. (10-36) reduces to

$$R' = E$$

Thus we have the following corollary:

COROLLARY 4*a*. To solve a Boolean equation such as Eq. (10-38), form the matrix E for the equation, and then the solution $f_1, \ldots, f_N$ is simply given by all unitary matrices R such that

$$R \rightarrow E \tag{10-42}$$

To solve simultaneous Boolean equations $E_1, E_2, \ldots, E_M$ we find unitary matrices R such that

$$R \rightarrow E_1 \cdot E_2 \cdot \cdots \cdot E_M$$

10-13. Application to Biochemical and Intelligence Problems

Logical Problems. In the examples that follow, the logical translation of English will be subordinated to the application of the matrix computational methods, once the logical transition has been accomplished. We shall be interested in the truth value of those sentences about which it is meaningful to say that they are *true* or *false*. Thus A_1 might mean "It was enzyme 1" is true; X_2 might mean "There is a dome radar" is true. The conjunction *and* corresponds to the logical operation $\cdot$, *or* (that is, the inclusive or) to $+$, *implies* to the relation $\rightarrow$, and *not*, or negation, to the complementation operation $^-$. The designation number corresponds to a truth table, where 1 means true and 0 means false.

The majority of logical problems involve given or accepted premises, facts, rules, or other logical relationships, which comprise the statement of "given functions" of the problems. There are two types of "solutions" to a set of given functions, the *antecedence* solutions and the *consequence* solutions. Antecedence solutions are *hypotheses* or *theories* from which the given functions can be deduced; consequence solutions can be *deduced from* the given functions. In other words, the truth of the antecedence

solutions is sufficient for the truth of the given functions, but the truth of the consequence solutions is necessary for the truth of the given functions. Finding anteeedence and consequence solutions is usually trivial; hence most logical problems require antecedence and consequence solutions of a *specified form*. Figure 10-13 illustrates this point and relates such problems to our Boolean matrix methods by means of the fundamental formulas.

Antecedence solutions — Given equations — Consequence solutions

$$F(f_1, \ldots, f_J, X_1, \ldots, X_K) \rightarrow E(A_1, \ldots, A_I, X_1, \ldots, X_K) \rightarrow F(f_1, \ldots, f_J, X_1, \ldots, X_K)$$

where $f_j = f_j(A_1, \ldots, A_I)$

In this case

$$\mathbf{F} \otimes \mathbf{R} \rightarrow \mathbf{E}$$
$$\mathbf{F}' = \overline{\bar{\mathbf{E}} \otimes \mathbf{R}^t}$$
$$\mathbf{R}' = \overline{\mathbf{F}^t \otimes \bar{\mathbf{E}}}$$

In this case

$$\mathbf{E} \rightarrow \mathbf{F} \otimes \mathbf{R}$$
$$\mathbf{F}^* = \mathbf{E} \otimes \mathbf{R}^t$$
$$\mathbf{R}^* = \overline{\mathbf{F}^t \otimes \mathbf{E}}$$

FIG. 10-13. Relationships between antecedence and consequence logical problems.

Analyzing Experimental Results. In biochemistry an enzyme often cannot be isolated easily, and therefore experiments performed with one enzyme frequently will involve several others. Consequently a combination of reactions must be observed. In these cases our logical computational methods can be extremely useful for evaluating experimental results and planning future experiments to yield the maximum information. Suppose that a chemist is studying enzymes A_1, A_2, A_3 in relation to reactions X_1, X_2, X_3. Suppose that he has done the following four experiments:

E_1: A solution containing none of A_1, A_2, A_3 produced reaction X_2, but not X_1 and not X_3.

E_2: The solution contained A_1 and either A_2 or A_3, or both—he could not be sure. The reaction was neither X_2 nor the combination of X_1 and X_3.

E_3: The solution had A_2 but not A_1, or else it did not have A_2 but had A_3. Reactions X_1 and X_2 occurred; or else reaction X_3 occurred, but X_1 did not.

E_4: The solution was obtained from a source that had A_3, and A_1 or A_2 or both, or else had neither A_1 nor A_3; the solution turned color, which meant that X_1 did not take place or that both X_2 and X_3 did.

In terms of symbolic logic these experiments can be interpreted thus:

$$E_1: \quad \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 = \bar{X}_1 \cdot X_2 \cdot \bar{X}_3$$
$$E_2: \quad A_1 \cdot (A_2 + A_3) = \bar{X}_2 \cdot \overline{(X_1 \cdot X_3)}$$
$$E_3: \quad \bar{A}_1 \cdot A_2 + \bar{A}_2 \cdot A_3 = X_1 \cdot X_2 + \bar{X}_1 \cdot X_3$$
$$E_4: \quad A_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot \bar{A}_3 = \bar{X}_1 + X_2 \cdot X_3$$

The problem is: What *theories* about the enzymes associated with each reaction will explain the experimental results? In other words we wish to find *antecedence* solutions of the form

$$F: \quad [X_1 = f_1(A_1, A_2, A_3)] \cdot [X_2 = f_2(A_1, A_2, A_3)] \cdot [X_3 = f_3(A_1, A_2, A_3)]$$

The first step is to determine the four designation-number matrices for the experiments:

$$E_1 = \begin{pmatrix} 0111 & 1111 \\ 0111 & 1111 \\ 1000 & 0000 \\ 0111 & 1111 \\ \\ 0111 & 1111 \\ 0111 & 1111 \\ 0111 & 1111 \\ 0111 & 1111 \end{pmatrix} \qquad E_3 = \begin{pmatrix} 1101 & 0001 \\ 1101 & 0001 \\ 1101 & 0001 \\ 0010 & 1110 \\ \\ 0010 & 1110 \\ 1101 & 0001 \\ 0010 & 1110 \\ 0010 & 1110 \end{pmatrix}$$

$$E_2 = \begin{pmatrix} 0001 & 0101 \\ 0001 & 0101 \\ 1110 & 1010 \\ 1110 & 1010 \\ \\ 0001 & 0101 \\ 1110 & 1010 \\ 1110 & 1010 \\ 1110 & 1010 \end{pmatrix} \qquad E_4 = \begin{pmatrix} 1010 & 0111 \\ 0101 & 1000 \\ 1010 & 0111 \\ 0101 & 1000 \\ \\ 1010 & 0111 \\ 0101 & 1000 \\ 1010 & 0111 \\ 1010 & 0111 \end{pmatrix}$$

Next, because of the special form of the solution F, we can utilize Corollary 4*a*:

$$R \rightarrow E_1 \cdot E_2 \cdot E_3 \cdot E_4 = \begin{pmatrix} 0000 & 0001 \\ 0001 & 0000 \\ 1000 & 0000 \\ 0000 & 1000 \\ \\ 0000 & 0100 \\ 0100 & 0000 \\ 0010 & 0010 \\ 0010 & 0010 \end{pmatrix}$$

The result is given by

f_1	0	1	1,0	1	1	0	1,0	0
f_2	1	0	1,1	0	1	0	1,1	0
f_3	0	1	1,1	0	0	1	1,1	0

There are thus four sets of solutions, one set for each of the four possibilities of X_1:

$$X_1 = f_1 = A_1 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3, \text{ or } \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot \bar{A}_2 \cdot A_3,$$
$$\text{or } \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot A_3, \text{ or } A_1 \cdot \bar{A}_3 + \bar{A}_1 \cdot \bar{A}_2 \cdot A_3$$
$$X_2 = f_2 = \bar{A}_1$$
$$X_3 = f_3 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$$

The first two sets (i.e., using the first two possibilities for X_1) are logically independent. The last two sets of solutions imply that the reactions are not independent but that $X_2 \cdot X_3 \rightarrow X_1$ and that $X_1 \cdot X_2 \cdot X_3 = 0$, respectively. These observations may lead to additional experiments for distinguishing between the four possibilities for X_1.

Analyzing Intelligence Reports. Suppose that the Blefuscudians' missile force has three kinds of intercontinental ballistic missile sites, which we shall denote by A_1, A_2, and A_3. Also, let X_1 represent their Hatlis missile, X_2 represent their dome radar, and X_3 represent their Bor missile. Suppose that military intelligence has determined that the distinguishing features which identify the three types of sites in terms of the equipment at the site are as follows:

$$E_1: \quad A_1 = X_1 \cdot \bar{X}_3 + X_2 \cdot X_3$$
$$E_2: \quad A_2 = X_2 \cdot \bar{X}_3 + \bar{X}_1 \cdot X_3$$
$$E_3: \quad A_3 = \bar{X}_1 \cdot \bar{X}_2 + X_2 \cdot X_3$$

These are the given information-function equations E. Now it happens that, on a high-altitude V-3 flight over Blefuscudia, observations of a certain area showed that there were either Hatlis missiles or Bor missiles or both, one could not be sure, and there was no dome radar—i.e., the observer reported $(X_1 + X_3) \cdot \bar{X}_2$. Suppose in addition it is known that a missile site of type A_2 never is in the same area as a site of type A_3, that is, $A_3 \rightarrow \bar{A}_2$. The problem then is: Assuming at least one missile site is present, what types of missile sites are there in the area? In more precise form, the problem is: What are the *consequence* solutions of the form F to the given intelligence information-function equation E?

$$F: \quad (X_1 + X_3) \cdot \bar{X}_2 = f(A_1, A_2, A_3)$$

To solve this problem, find

$$F = \begin{pmatrix} 10 \\ 01 \\ 10 \\ 10 \\ \\ 01 \\ 01 \\ 10 \\ 10 \end{pmatrix} \qquad E = \begin{pmatrix} 0000 & 1000 \\ 0100 & 0000 \\ 0010 & 0000 \\ 0001 & 0000 \\ \\ 0000 & 0010 \\ 1000 & 0000 \\ 0000 & 0001 \\ 0000 & 0100 \end{pmatrix}$$

whence from the right-hand form of Eqs. (10-36) we have

$$R = \overline{\bar{F}^t \otimes E} = \begin{pmatrix} 0011 & 1101 \\ 1100 & 0010 \end{pmatrix}$$

Thus $f = (1100 \quad 0010)$

However, at least one type of site must be in the area—that is, $A_1 + A_2 + A_3$ is a constraint; similarly $A_3 \rightarrow \bar{A}_2$ is treated as a constraint. Now the conditions corresponding to the zeros of the designation number of a constraint cannot occur. Thus the solution is given by

$$\begin{aligned}(A_1 + A_2 + A_3) \cdot (A_3 \rightarrow \bar{A}_2) \cdot (f) &= (0111 \quad 1111) \cdot (1111 \quad 1100) \cdot (1100 \quad 0010) \\ &= (0100 \quad 0000) \\ &= A_1 \cdot \bar{A}_2 \cdot \bar{A}_3\end{aligned}$$

In other words, from the observations, together with the given intelligence information, it can be deduced that only an A_1 missile site is in the area.

10-14. Exercises

Suppose that A is false, B is true, and C is false; determine the *truth value* of the following combined elements:

	Answer
10-1. $\bar{A} \cdot B \cdot \bar{C}$	True
10-2. $\bar{A} \cdot B + A \cdot \bar{B}$	True
10-3. $(\bar{A} \cdot B + A \cdot \bar{B}) \cdot C$	False
10-4. $\overline{A \cdot C}$	True
10-5. $\overline{A + B}$	False
10-6. $(A + B) \cdot C + (A + C) \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot \bar{C}$	False

10-7. Note that $A \rightarrow B$ is true if both A is false and B is false. Can you think of examples of this situation that might occur in real life?

10-8. Consider the sentence "$A + \bar{A}$ is a tautology." Is this tautologically true, or is it factually true?

10-9. Using $I \cdot A = A$ and (8) and (15) of Sec. 10-3, prove (18).

Solution

$$\begin{aligned}A + \bar{A} \cdot B &= (A + \bar{A}) \cdot (A + B) = I \cdot (A + B) \\ &= A + B\end{aligned}$$

as desired.

10-10. What other equations listed in Sec. 10-3 would be used to prove (19)?

Solution. (1), (2), (3), (6), (7), (8). [HINT: Start with the right-hand side of (19) and apply (8), using (1), (2), (3), (6), and (7) to simplify the terms.]

Are the following tautologies?

	Answer
10-11. $\bar{A} \cdot B \rightarrow A$	No
10-12. $A \rightarrow A + B$	Yes
10-13. $(A \rightarrow B) = (\bar{B} \rightarrow \bar{A})$	Yes
10-14. $(A \cdot B \cdot C \rightarrow D) = (\bar{D} \cdot B \cdot C \rightarrow \bar{A})$	Yes

10-15. Define I and 0 in terms of p_2 as given in Sec. 10-3.

10-16. Show that the following is a tautology:

$$p_1[p_1(x,p_1\{y,z\}),p_1(p_1\{u,p_1[u,u]\},p_1\{p_1[v,y],p_1[p_1(x,v),p_1(x,v)]\})]$$

10-17. Show by set diagrams that $A + B \cdot C = (A + B) \cdot (A + C)$.

If a point is in set A, not in set B, and not in set C, then is it in the following sets, i.e., in the sets for which the following combined elements are true?

	Answer
10-18. $(A + B) \cdot \bar{C}$	Yes
10-19. $A + B \cdot \bar{C}$	Yes
10-20. $A \cdot B \cdot \bar{C}$	No
10-21. $A \cdot (B + \bar{C})$	Yes
10-22. $\bar{B} + A \cdot C$	Yes

10-23. Draw the set diagram for $A \rightarrow B$.

10-24. Draw the set diagram for $A = B$.

Find the truth tables for:

10-25. $(\overline{A \cdot B}) + A$. **10-26.** $\overline{A \cdot C}$. **10-27.** $\bar{A} \cdot B \cdot \bar{C}$.

10-28. $\overline{A + B}$. **10-29.** $(\bar{A} \cdot B + A \cdot \bar{B}) \cdot \overline{C}$.

10-30. Show, by means of truth tables, that $\overline{A \cdot B} + (A + \bar{B}) \cdot (\bar{A} + B)$ and $A + \bar{A}$ are tautologies.

With respect to standard bases determine the following:

		Answer			
10-31.	$\#(A + \bar{B})$	1101			
10-32.	$\#(A \cdot \bar{B})$	0100			
10-33.	$\#(A \cdot \bar{B} + \bar{C})$	1111	0100		
10-34.	$\#[A \cdot (\bar{B} + \bar{C})]$	0101	0100		
10-35.	$\#[A \cdot B + \bar{A} \cdot \bar{B}]$	1001			
10-36.	$\#[A \cdot \bar{B} + \bar{A} \cdot B]$	0110			
10-37.	$\#[A \cdot \bar{B} + C \cdot \bar{D}]$	0100	1111	0100	0100
10-38.	$\#[A \cdot (B + \bar{C}) + \bar{A} \cdot (\bar{B} + C)]$	1101	1011		
10-39.	$\#[A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}]$	1000	0001		
10-40.	$\#[(A + B + C) \cdot (\bar{A} + \bar{B} + \bar{C})]$	0111	1110		
10-41.	$\#[A \cdot \bar{B} \cdot C + \bar{C} \cdot (D + B \cdot \bar{E}) + \bar{D} \cdot E]$	0011	0100	1111	0100
			1111	1111	1111 0100

Find the first and second canonical form corresponding to the following designation numbers:

		Answer
10-42. 1001	0110	$\#[\bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C]$ $\#[(\bar{A} + B + C) \cdot (A + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C})]$
10-43. 0101	0101	$\#[A \cdot B \cdot C + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot \bar{C}]$ $\#[(A + B + C) \cdot (A + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (A + \bar{B} + \bar{C})]$
10-44. 1000	0001	$\#[\bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot C]$ $\#[(\bar{A} + B + C) \cdot (A + \bar{B} + C) \cdot (\bar{A} + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + B + \bar{C}) \cdot (A + \bar{B} + \bar{C})]$

10-45. 0111	1110			$\#[A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C]$ $\#[(A + B + C) \cdot (\bar{A} + \bar{B} + \bar{C})]$
10-46. 1011				$\#[\bar{A} \cdot \bar{B} + \bar{A} \cdot B + A \cdot B]$ $\#[\bar{A} + B]$
10-47. 0010				$\#(\bar{A} \cdot B)$ $\#[(A + B) \cdot (\bar{A} + B) \cdot (\bar{A} + \bar{B})]$
10-48. 1011	0001	0110	0101	$\#[\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot C \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot D + A \cdot B \cdot C \cdot D]$ $\#[(\bar{A} + B + C + D) \cdot (A + B + \bar{C} + D) \cdot (\bar{A} + B + \bar{C} + D) \cdot (A + \bar{B} + \bar{C} + D) \cdot (A + B + C + \bar{D}) \cdot (\bar{A} + \bar{B} + C + \bar{D}) \cdot (A + B + \bar{C} + \bar{D}) \cdot (A + \bar{B} + \bar{C} + \bar{D})]$

Write down directly mongrel forms for:

				"Good" solution
10-49. 0110	1001			$(A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$
10-50. 1001	0110	0110	1001	$(A \cdot B + \bar{A} \cdot \bar{B}) \cdot (C \cdot D + \bar{C} \cdot \bar{D}) + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot (C \cdot \bar{D} + \bar{C} \cdot D)$
10-51. 0010	0100	0010	0010	$(\bar{A} \cdot B) \cdot (\bar{C} + D) + (A \cdot \bar{B}) \cdot (C \cdot \bar{D})$
10-52. 1010	0010	0010	0010	$\bar{A} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B$
10-53. 0111	1011	1101	1110	$(A + B) \cdot \bar{C} \cdot \bar{D} + (\bar{A} + B) \cdot C \cdot \bar{D} + (A + \bar{B}) \cdot \bar{C} \cdot D + (\bar{A} + \bar{B}) \cdot C \cdot D$
10-54. 1011	0111	1011	0111	$(\bar{A} + B) \cdot (\bar{C} \cdot \bar{E}) + (A + B) \cdot C \cdot \bar{E}$
1111	0000	1111	1111	$+ (\bar{C} + D) \cdot E$

Find, by the method of Sec. 10-7, the simplest sum-of-products forms of the following designation numbers:

				Solution
10-55. 0111	0101			$A + B \cdot \bar{C}$
10-56. 0110	1001			$A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot C$
10-57. 0100	0100	1111	0100	$A \cdot \bar{B} + \bar{C} \cdot D$
10-58. 0010	0100	0010	0010	
10-59. 0011	1010	1001	0010	1000 1101 0100 1011

The *simplest product-of-sums* form is a product of sums that has the least number of operations + and · . Find the simplest product-of-sums forms of each of the following designation numbers. (HINT: Work as in Sec. 10-7, but with the zeros instead of the units.)

				Solution
10-60. 1001	1010			$(\bar{A} + B) \cdot (A + \bar{B} + C) \cdot (\bar{A} + \bar{C})$
10-61. 1111	0011	0001	1000	$(B + \bar{C} + D) \cdot (B + C + \bar{D}) \cdot (A + \bar{B} + \bar{D}) \cdot (\bar{A} + \bar{C} + \bar{D})$
10-62. 0100	1001	1101	0110	
10-63. 0110	0101	1001	1010	
10-64. 0001	0011	0111	1111	

Find the simplest sum-of-products forms of the following designation numbers by the method of Sec. 10-8:

10-65. 0111 0101.
10-66. 0110 1001.
10-67. 0100 0100 1111 0100.
10-68. 0010 0100 0010 0010.
10-69. 0011 1010 1001 0010 1000 1101 0100 1011.

Find the *simplest product-of-sums* forms of each of the following designation numbers, using the method of Sec. 10-8, but working with the zeros instead of the units:

10-70. 1001 1010.
10-71. 0100 1001 1101 0110.
10-72. 1111 0011 0001 1000.
10-73. 0110 0101 1001 1010.
10-74. 0001 0011 0111 1111.

Find the constrained basis for the following sets of constraints:

Answer

10-75. $A \cdot B = 0$

$\#A = 010$
$\#B = 001$

10-76. $B \cdot C = 0$

$\#A = 0101 \quad 01$
$\#B = 0011 \quad 00$
$\#C = 0000 \quad 11$

10-77. $\bar{A} + B \cdot C = 0$
$A \cdot C = 0$

$\#A = 11$
$\#B = 01$
$\#C = 00$

10-78. $A \rightarrow C$
$C \rightarrow B$

$\#A = 0001$
$\#B = 0111$
$\#C = 0011$

10-79. $A \cdot (B \cdot \bar{C} + \bar{B} \cdot C) = 0$
$A \cdot B \rightarrow 0$
$B + \bar{D} = 1$

$\#A = 0100 \quad 000$
$\#B = 0010 \quad 111$
$\#C = 0001 \quad 101$
$\#D = 0000 \quad 011$

Find the constraints corresponding to the following constrained bases:

Answer

10-80. $\#A = 010$
$\#B = 001$

$\bar{A} + \bar{B} = I$

10-81. $\#A = 111$
$\#B = 010$
$\#C = 001$

$A \cdot \bar{B} + A \cdot \bar{C} = I$

10-82. $\#A = 0001$
$\#B = 0111$
$\#C = 0011$

$\bar{A} \cdot \bar{C} + B \cdot C = I$

With respect to the constrained basis given in Sec. 10-9, find the designation numbers of:

Answer

10-83. $A \cdot \bar{B} + \bar{C}$ — 1111 01
10-84. $A \cdot (\bar{B} + \bar{C})$ — 0101 01
10-85. $A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}$ — 1000 00
10-86. $A \cdot (B + \bar{C}) + \bar{A} \cdot (\bar{B} + C)$ — 1101 10

With respect to the constrained basis given in Sec. 10-9, find the simplest sum-of-products forms of:

Answer

10-87. 0101 01 — A
10-88. 0111 01 — $A + B$
10-89. 1010 01 — $\bar{A} \cdot \bar{C} + A \cdot C$
10-90. 0101 0100 — A
10-91. 0111 0101 — $A + B$
10-92. 1010 0101 — $\bar{A} \cdot \bar{C} + A \cdot C$

Are the following sets of functions independent? If not, find the logical dependence.

Answer

10-93. $A \cdot B + \bar{A} \cdot \bar{B}$
$A \cdot \bar{B} + \bar{A} \cdot B$ — No

10-94. $\bar{A}$
$\bar{B}$ — Yes

10-95. $\bar{B}$
$\bar{A}$
$(A \cdot B + \bar{A} \cdot \bar{B}) \cdot \bar{C} + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot C$ — Yes

10-96. $\bar{B} \cdot \bar{C} + \bar{A} \cdot C$
$\bar{A} \cdot \bar{C} + \bar{B} \cdot C$
$(A + \bar{B}) \cdot \bar{C} + A \cdot \bar{B} \cdot C$ — Yes

10-97. $\bar{B} \cdot \bar{C} + \bar{A} \cdot C$
$\bar{A} \cdot \bar{C} + \bar{B} \cdot C$
$\bar{B}$ — No

Prove the following useful properties of Boolean matrices:

10-98. $(A \otimes B) \otimes C = A \otimes (B \otimes C) = A \otimes B \otimes C$.
10-99. $(A + B) + C = A + (B + C) = A + B + C$.
10-100. $(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C$.
10-101. $(A \otimes B)^t = B^t \otimes A^t$.
10-102. If $A \rightarrow C$ and $B \rightarrow D$, then $A \otimes B \rightarrow C \otimes D$.
10-103. $\overline{A^t} = \bar{A}^t$.
10-104. $\overline{A \cdot B} = \bar{A} + \bar{B}$, $\overline{A + B} = \bar{A} \cdot \bar{B}$.
10-105. If $A \rightarrow B$ then $\bar{B} \rightarrow \bar{A}$.

Find the Boolean transformations that will transform F into E:

10-106. $F = (f_1 + f_3) \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3 + \bar{f}_1 \cdot \bar{f}_2 \cdot X_1 \cdot \bar{X}_2 \cdot \bar{X}_3 + (f_2 + f_3) \cdot \bar{X}_1 \cdot X_2 \cdot \bar{X}_3$
$+ f_1 \cdot X_1 \cdot X_2 \cdot \bar{X}_3 + \bar{f}_1 \cdot \bar{f}_2 \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot X_3 + f_1 \cdot f_2 \cdot X_1 \cdot \bar{X}_2 \cdot X_3$
$+ f_2 \cdot \bar{X}_1 \cdot X_2 \cdot X_3 + (\bar{f}_1 \cdot \bar{f}_2 + f_2 \cdot f_1) \cdot X_1 \cdot X_2 \cdot X_3$.
$E = ((\bar{A}_1 + A_2) \cdot \bar{A}_3 + A_3 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2)) \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3$
$+ (A_2 \cdot \bar{A}_3 + A_1 \cdot A_2 \cdot A_3) \cdot X_1 \cdot \bar{X}_2 \cdot \bar{X}_3$
$+ ((A_1 + A_2) \cdot \bar{A}_3 + A_3 \cdot \bar{A}_1) \cdot \bar{X}_1 \cdot X_2 \cdot \bar{X}_3$
$+ (\bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + A_3 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2)) \cdot X_1 \cdot X_2 \cdot \bar{X}_3$
$+ (A_2 \cdot \bar{A}_3 + A_1 \cdot A_2 \cdot A_3) \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot X_3 + \bar{A}_1 \cdot A_2 \cdot A_3 \cdot X_1 \cdot \bar{X}_2 \cdot X_3$
$+ (A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3) \cdot \bar{X}_1 \cdot X_2 \cdot X_3 + A_2 \cdot X_1 \cdot X_2 \cdot X_3$.

Solution. $f_1 = \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + A_3 \cdot (\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_2), f_2 = A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3$, $f_3 = A_2 \cdot \bar{A}_3$.

10-107. $F = \bar{X}_1 \cdot \bar{X}_2 \cdot f_1 + X_1 \cdot \bar{X}_2 \cdot f_3 + \bar{X}_1 \cdot X_2 \cdot f_2 + X_1 \cdot X_2 \cdot (f_2 + f_1 \cdot \bar{f}_3)$

$E = \bar{X}_1 \cdot \bar{X}_2 \cdot (\bar{A}_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) + A_3 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2))$
$+ X_1 \cdot \bar{X}_2 \cdot \bar{A}_1 + \bar{X}_1 \cdot X_2 \cdot (\bar{A}_3 \cdot (A_1 + A_2) + A_3 \cdot (\bar{A}_1 + A_2))$
$+ X_1 \cdot X_2 \cdot (\bar{A}_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) + A_1 \cdot \bar{A}_2 \cdot A_3).$

10-108. $F = \bar{X}_1 \cdot \bar{X}_2 \cdot (f_2 \cdot \bar{f}_3 + f_1 \cdot \bar{f}_2 \cdot f_3) + X_1 \cdot \bar{X}_2 \cdot (\bar{f}_1 \cdot f_2 \cdot \bar{f}_3 + f_3 \cdot (\bar{f}_1 + f_2))$
$+ \bar{X}_1 \cdot X_2 \cdot (f_1 \cdot \bar{f}_2 \cdot \bar{f}_3 + \bar{f}_2 \cdot f_3) + X_1 \cdot X_2 \cdot (\bar{f}_1 \cdot \bar{f}_2 \cdot \bar{f}_3 + f_1 \cdot f_2).$

$E = \bar{X}_1 \cdot \bar{X}_2 \cdot (\bar{A}_1 \cdot A_2 \cdot \bar{A}_3 + A_2 \cdot A_3)$
$+ X_1 \cdot \bar{X}_2 \cdot (A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + A_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2))$
$+ \bar{X}_1 \cdot X_2 \cdot (\bar{A}_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) + A_3 \cdot \bar{A}_2 \cdot \bar{A}_1)$
$+ X_1 \cdot X_2 \cdot (\bar{A}_3 \cdot A_2 \cdot \bar{A}_1 + A_3 \cdot (\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_2)).$

Solve the following sets of simultaneous Boolean equations for the unknowns X and Y:

10-109. $(A + B) \cdot X + A \cdot \bar{X} \cdot \bar{Y} = A$, $\bar{A} \cdot \bar{Y} + X = A + B$, $X \cdot \bar{Y} \rightarrow \bar{B}$.

Solution. $X = A$, $Y = A \cdot B + \bar{A} \cdot \bar{B}$; $X = A$, $Y = A + \bar{B}$.

10-110. $X \cdot Y = A \cdot \bar{B}$, $A + B = \bar{A} \cdot Y + X$, $(A + B) \cdot X + A \cdot Y = A$.

Solution. $X = A$, $Y = A \cdot \bar{B} + \bar{A} \cdot B$.

10-111. $X \cdot Y = A \cdot \bar{B}$, $\bar{A} \cdot B = \bar{A} \cdot Y + X$, $(A + B) \cdot X + A \cdot Y = A$.

No solution. Why?

10-112. $X \cdot (A + B) + Y \cdot (A + C) = (\bar{X} \cdot \bar{Y} + X \cdot Y) \cdot (\bar{A} + \bar{B} + \bar{C})$,
$X \cdot (A \cdot B + \bar{A} \cdot \bar{B}) \rightarrow Y + A \cdot B + C.$

There are four sets of solutions:

$X = A \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$ $\quad$ $X = A \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$
$Y = \bar{C} + A \cdot \bar{B} + \bar{A} \cdot B$ $\quad$ $Y = \bar{C} + (\bar{A} + \bar{B}) \cdot C$

$X = (A + B) \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$ $\quad$ $X = (A + B) \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$
$Y = \bar{C} + A \cdot \bar{B} + \bar{A} \cdot B$ $\quad$ $Y = \bar{C} + (\bar{A} + \bar{B}) \cdot C$

10-113. Solve the following set of simultaneous implication equations:

1. $(B + \bar{A} + D + C) \cdot \bar{Y} \rightarrow \bar{X}$
2. $A \cdot \bar{B} + \bar{A} \cdot B \rightarrow \overline{X \cdot Y}$
3. $X + Y \rightarrow A \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} \cdot \bar{D}$
4. $A \cdot \bar{C} \cdot D + B \cdot \bar{C} \cdot \bar{D} \rightarrow Y$
5. $A \cdot B \cdot \bar{C} \cdot Y \rightarrow X$

There are three pairs of solutions to this set of five implications:

$X = A \cdot \bar{C} \cdot (B + \bar{D})$ $\quad$ $X = A \cdot B \cdot \bar{C}$ $\quad$ $X = A \cdot B \cdot \bar{C}$
$Y = \bar{C} \cdot (A \cdot D + B \cdot \bar{D})$ $\quad$ $Y = \bar{C} \cdot (A + B \cdot \bar{D})$ $\quad$ $Y = \bar{C} \cdot (A \cdot D + B \cdot \bar{D})$

10-114. Find $f_1 = f_1(A_1, A_2)$, $f_2 = f_2(A_1, A_2)$, where

$h_1 = A_2 + A_3$ $\quad$ $g_1 = h_1 \cdot h_2$ $\quad$ $f_1 = g_1 + g_2$
$h_2 = A_1 + A_3$ $\quad$ $g_2 = \bar{h}_1 \cdot \bar{h}_2$ $\quad$ $f_2 = g_2 + g_3$
$\quad$ $g_3 = h_1 \cdot \bar{h}_2$

10-115. Find $f_1 = f_1(g_1, g_2, g_3)$, $f_2 = f_2(g_1, g_2, g_3)$ so that the following relations will be satisfied:

$h_1 = A_2 + A_3$ $\quad$ $g_1 = h_1 \cdot h_2$
$h_2 = A_1 + A_3$ $\quad$ $g_2 = \bar{h}_1 \cdot \bar{h}_2$
$\quad$ $g_3 = h_1 \cdot \bar{h}_2$

$F = (\bar{f}_1 + f_2) \cdot X_1 \cdot \bar{X}_2 + (f_1 + f_2) \cdot \bar{X}_1 \cdot X_2 + (f_1 \cdot f_2 + \bar{f}_1 \cdot \bar{f}_2) \cdot \bar{X}_1 \cdot \bar{X}_2$
$E = (\bar{A}_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot \bar{A}_3 \cdot X_1) \cdot \bar{X}_2 + (\bar{A}_1 + A_2 + A_3) \cdot \bar{X}_1 \cdot X_2$

10-116. *All Solutions to the Boolean Matrix Equation.* We introduce a new kind of matrix-multiplication operation, called the *theta* product, defined as follows: $(S_{ji}) = (P_{jk}) \ \Theta \ (Q_{ki})$ if and only if for each position

$$S_{ji} = \prod_k (P_{jk} = Q_{ki})$$
$$= \prod_k (P_{jk} \cdot Q_{ki} + \bar{P}_{jk} \cdot \bar{Q}_{ki})$$

Show that given the matrix equation

$$(a_{ik}) \otimes (x_{kj}) = (b_{ij})$$

—where $i = 0, 1, \ldots, I$; $j = 0, 1, \ldots, J$; and $k = 0, 1, \ldots, K$—if

$$(S_{mj}) = \{(a_{ik}) \otimes (b[X_0, X_1, \ldots, X_K])\}^t \ \Theta \ (b_{ij})$$

then when (S_{mj}) is interpreted as an R-type matrix, all solutions to the matrix equation are found.

REFERENCES

For detailed discussions of the formal theory of symbolic logic, the reader is referred to the following:

Church, A.: Introduction to Mathematical Logic, Part I, *Ann. Math. Studies* 13, Princeton University Press, Princeton, N.J., 1944.

Cohen, M., and E. Nagel: "An Introduction to Logic and the Scientific Method," Harcourt, Brace and Company, Inc., New York, 1934.

Hilbert, D., and W. Ackerman: "Principles of Mathematical Logic," R. E. Luce (ed.), Chelsea Publishing Company, New York, 1950.

Quine, Willard van Orman: "Mathematical Logic," rev. ed., Harvard University Press, Cambridge, Mass., 1951.

Reichenbach, H.: "Elements of Symbolic Logic," The Macmillan Company, New York, 1950.

Rosser, J. B.: "Logic for Mathematicians," McGraw-Hill Book Company, Inc., New York, 1953.

Whitehead, A. N., and B. Russell: "Principia Mathematica," vol. 1, 1910, vol. 2, 1912, vol. 3, 1913, Cambridge University Press, New York (2d ed., 1925–1927).

The following references deal with the foundations and the axiomatic approach to Boolean algebras:

Bernstein, B. A.: A Set of Four Independent Postulates for Boolean Algebra, *Trans. Am. Math. Soc.*, vol. 17, 1916; Sets of Postulates for the Logic of Propositions, *Trans. Am. Math. Soc.*, vol. 28, 1926.

Boole, G.: "George Boole's Collected Logical Works" (1859), vols. 1 and 2, P. E. B. Jourdain (ed.), The Open Court Publishing Company, La Salle, Ill., 1940; "An Investigation of the Laws of Thought," Dover Publications, Inc., New York, 1951.

Copilowish, I. M.: Matrix Development of the Calculus of Relations, *J. Symbolic Logic*, vol. 13, no. 4, pp. 193–203, December, 1948.

Huntington, E. V.: Sets of Independent Postulates for the Algebra of Logic, *Trans. Am. Math. Soc.*, vol. 5, pp. 288–309, 1904; A New Set of Independent Postulates for the Algebra of Logic, *Proc. Natl. Acad. Sci. U.S.*, vol. 18, 1932; New Sets of Independent Postulates for the Algebra of Logic, *Trans. Am. Math. Soc.*, vol. 35, 1933.

Jevons, W. S.: "Pure Logic (1864) and Other Minor Works," R. Adamson and H. A. Jevons (eds.), The Macmillan Company, London and New York, 1890.

Schroeder, E.: "Algebra der Logik," vol. 1, Teubner Verlagsgesellschaft, Leipzig, 1890.

A discussion of the axiomatic method and techniques for proving the independence of and equivalence between axiom systems is found in R. L. Wilder, "Introduction to the Foundations of Mathematics," John Wiley & Sons, Inc., New York, 1952.

For further discussion of the logical translation of English see:

Allen, L. E.: Toward More Clarity in Business Communications by Modern Logical Methods, *Management Sci.*, vol. 5, no. 1, pp. 121–135, October, 1958.

Carnap, R.: "The Logical Syntax of Language," Routledge and Kegan Paul, Ltd., London, 1949.

Ledley, R. S.: Mathematical Foundations and Computational Methods for a Digital Logic Machine, *J. Operations Research Soc. Am.*, vol. 2, no. 3, pp. 249–274, August, 1954. (See also letters on Logical Translation of English by L. Davidson, *J. Operations Research Soc. Am.*, vol. 3, no. 2, p. 466, November, 1954, and J. W. Gilmore, *J. Operations Research Soc. Am.*, vol. 3, no. 1, p. 104, February, 1955.)

Reichenbach, H.: "Elements of Symbolic Logic," The Macmillan Company, New York, 1947.

References on Simplification of Boolean Functions

Abhyankar, S.: Minimal "Sum of Products of Sums" Expressions of Boolean Functions, *IRE Trans. on Electronic Computers*, vol. EC-7, no. 4, pp. 268–276, December, 1958.

Harris, B.: An Algorithm for Determining Minimal Representation of a Logic Function, *IRE Trans. on Electronic Computers*, vol. EC-6, no. 2, pp. 103–108, June, 1957.

Karnaugh, G.: The Map Method for Synthesis of Combinational Logic Circuits, *Trans. AIEE*, pt. I, *Communs. and Electronics*, vol. 72, pp. 593–599, November, 1953.

Ledley, R. S.: Mathematical Foundations and Computational Methods for a Digital Logic Machine, *J. Operations Research Soc. Am.*, vol. 2, no. 3, pp. 249–279, August, 1954.

McCluskey, E. J., Jr.: Minimization of Boolean Functions, *Bell System Tech. J.*, vol. 35, pp. 1–28, November, 1956.

Quine, W. v. O.: A Way to Simplify Truth Functions, *Am. Math. Monthly*, vol. 62, pp. 627–631, November, 1955.

Roth, J. P.: Algebraic Topological Methods for the Synthesis of Switching System I, *Trans. Am. Math. Soc.*, vol. 88, no. 2, pp. 301–326, July, 1958.

Veitch, E. W.: A Chart Method for Simplifying Truth Functions, *Proc. Assoc. Computing Mach.*, May, 1952, pp. 127–133.

CHAPTER 11

SEARCHING, SORTING, ORDERING, AND CODIFYING

11-1. Introduction

Ingredients and Definitions. The ingredients of searching are *items* and *characteristics* and *the associations of characteristics with items.* An item is a package of information; the characteristics associated with this item are attributes that are related to the item. Problems of searching all involve locating an item, or items, associated with a given set of characteristics. For example, each article reference included in a bibliography is an item; the subjects relating to the contents of the article are the characteristics. A bill of sale is an item; the name and address of the buyer may be considered as the characteristics. A research grant is an item; the name of the principal investigator, the amount of the grant, the subject under investigation, the name of the school or organization receiving the grant, etc., are all characteristics. The inventories of particular replacement parts are the items; the serial numbers, costs, acquisition record, etc., are the characteristics—and so forth.

Given a set of characteristics, the process of obtaining *every item associated with all the given characteristics* is called *searching.* In a bibliographical list of research reports one might search for all articles concerning the application of nuclear theory.† The words *application*, *nuclear*, and *theory* are the characteristics, and the search should result in a collection of references, each concerned with the application of nuclear theory. As a specific illustration, consider the *item list* of article references and the *characteristic list* of subjects of a sample bibliography given in Fig. 11-1. We have given each item a boldface number and each characteristic an italicized number. The *item-characteristic associations* are given in Table 11-1, where a unit indicates that the characteristic of the row of the unit is associated with the item of its column. Consider the rows

† In this chapter we neglect any theory of semantics or meaning associated with the characteristics. For this important problem see, for example, Y. Bar-Hillel, Some Theoretical Aspects of the Mechanization of Literature Searching, *Tech. Rept.* 3, ONR Contract No. N62558-2214, April, 1960; and A Logician's Reaction to Recent Theorizing on Information Search Systems, *Am. Document.*, vol. 8, no. 2, pp. 113–113, April, 1957 (with comments, pp. 103–122).

PART I. THE ARTICLE LIST (BIBLIOGRAPHY PROPER)

Article No.		*Article No.*		*Article No.*	
1.1	Abrahams, A. P. Autoradiographic determination of radioactivity in rocks. *Nucleonics* 15:85–86 Mar 1957	**2.1**	Nicholls, J. Alpha-scintillation monitor for hands and clothing. *Nucleonics* 15:80, 81, 83, 84 Mar 1957	**3.1**	Senior, D. A. The Kerr cell, a high speed electro-shutter, Pt. II. *Instr. Pract.* 11:471–476 May 1957
1.2	Aravindakshan, C. A simple arrangement for obtaining optical transforms of crystal structures. *J. Sci. Instr.* 34:250 Jn 1957	**2.2**	Pope, M. I. An automatically recording vacuum balance. *J. Sci. Instr.* 34:229–232 Jn 1957	**3.2**	Smith, B. O., and Grimshaw, A. G. A pneumatic level indicator. *Instr. Pract.* 11:469–470 May 1957
1.3	Gasstrom, R. V. A very fast pulse-height analyzer with independent uptake, sorting, and storage of information. *Nuclear Instruments* 1:75–79 Mar 1957	**2.3**	Powell, D. A. An apparatus giving thermogravimetric and differential thermal curves simultaneously from one sample. *J. Sci. Instr.* 34:225–227 Jn 1957	**3.3**	Stockendal, R., and Bergkvist, K. E. Evaporation device for beta-spectrometer samples. *Nuclear Instruments* 1:53–54 Jan 1957
1.4	NBS Circular 850. Bibliography on ignition and spark ignition systems. Nov 1 1956	**2.4**	Seidle, F. G. P., et al. Modification of the Brookhaven fast chopper. *Nuclear Instruments* 1:92–93 Mar 1957	**3.4**	Tove, Per-Arne. Electronic time analyzer applied to the measurement of the half-lives of metastable nuclear states. *Nuclear Instruments* 1:95–100 Mar 1957

PART II. THE WORD LIST (CODE BOOK AND THESAURUS)

	Word No.		*Word No.*		*Word No.*
adaptability	*3.2*	evaluation	*6.1*	Netherlands	*4.2*
analysis	*4.1*	gas	*2.2*	nuclear	*4.3*
application	*5.1*	heat	*2.3*	plan	*6.2*
concept	*6.2*	hysteresis	*1.1*	theory	*4.4*
counting	*2.1*	implementation	*5.3*	thermal	*2.3*
design	*7.1*	instrumentation	*5.3*	use	*5.1*
differentiation	*3.1*	mass	*1.2*	versatility	*3.2*
England	*5.2*				

FIG. 11-1. Illustrative bibliography example.

associated with the given characteristics. Form a new row that has a unit corresponding only to those (column) positions where all the given characteristic rows also have units, i.e., *logically multiply* these rows.

TABLE 11-1. ASSOCIATION OF ITEMS WITH CHARACTERISTICS

Characteristic	Item											
	1.1	**1.2**	**1.3**	**1.4**	**2.1**	**2.2**	**2.3**	**2.4**	**3.1**	**3.2**	**3.3**	**3.4**
1.1	0	0	0	0	0	0	0	0	0	0	0	1
1.2	0	0	0	0	0	1	0	0	0	0	0	0
2.1	1	0	0	0	0	0	0	1	0	0	0	0
2.2	0	0	0	1	0	1	0	0	0	0	0	0
2.3	0	0	0	1	0	0	1	0	0	0	0	0
3.1	0	0	0	0	0	1	1	0	0	0	0	1
3.2	1	0	0	0	0	0	1	0	0	0	0	1
4.1	0	0	1	0	0	1	0	0	0	0	1	1
4.2	0	0	1	0	0	0	0	1	0	0	1	1
4.3	0	0	0	0	1	0	0	1	0	0	1	1
4.4	0	0	1	0	0	0	0	1	0	0	1	1
5.1	0	0	0	0	0	1	1	1	0	1	0	1
5.2	0	1	0	0	0	1	1	0	1	1	0	0
5.3	0	0	0	1	0	0	1	1	1	0	1	0
6.1	1	1	1	0	0	1	0	0	1	0	1	0
6.2	0	1	0	0	1	0	0	1	1	1	1	0
7.1	0	1	0	1	1	1	1	0	1	1	0	0

The units of the resulting row correspond to the columns of the desired items. For example, given *application* (*5.1*), *nuclear* (*4.3*), *theory* (*4.4*), we logically multiply rows,

	5.1	0000	0111	0101
	4.3	0000	1001	0011
	4.4	0010	0001	0011
obtaining		0000	0001	0001

with units in columns **3.4** and **2.4** corresponding to items by Tove and Seidle. However, it is rarely feasible to display a matrix such as in Table 11-1; we have used it simply as an illustration to aid the visualization of the searching methods to be described.

It is difficult to discuss searching without using the terms *sorting*, *ordering*, and *codifying*. It is often helpful for temporary use to substitute a symbolic abbreviation to represent an item or characteristic.

For example, the use of boldface and italicized numbers in our illustrative bibliography made the table of associations easier to display. *A code is a list of such abbreviations; the making of a code is codifying.*† Frequently the code is given in the form of a "dictionary," or "thesaurus." In our illustrative bibliography the code for the item (article) list is presented in dictionary form, and the code numbers are listed in numerical order. The characteristic (word) list is in thesaurus form, and the word list is in alphabetical order; here synonyms are given the same code number. Codes are used in searching as follows: First the thesaurus of characteristics is compared with the list of desired characteristics, and the corresponding code numbers are found; next the search is made to find the code numbers of the resulting items; finally the actual items are determined from the dictionary. In computer searching the use of codes in this manner is almost always indicated. Further discussion of codes appears in Sec. 11-7.

By *ordering a list* is meant the process of arranging the list according to some linear ordering system, such as alphabetically or numerically, so that, if element Q of the list is greater than element P, then element Q lies further from the beginning of the list than P. By *sorting a list* is meant the process of separating the list into partitions, i.e., *mutually exclusive classes.* In sorting items the *equivalence relation* by means of which the classes, or partitions, are defined is often given in terms of associated characteristics. For example, research grants may be sorted according to the university that received the grant, etc. Searching may be defined as the process of sorting a list of items into two categories, those items which are associated at least with all of certain given characteristics, and those which are not. Frequently the classes, or partitions, in a sort are ordered. For example, collections of research grants sorted by university can be ordered alphabetically by the university's names.

Searching and sorting are both closely associated with ordering. For instance, to sort research grants by university, all that need be done is to order the grants alphabetically by university, and then the result will present all grants from the same university juxtaposed in the list; hence the grants will automatically be sorted as the partitions of the sort are ordered.

Summarizing, to *sort* a list of elements, some equivalence relation must be defined that tells when two elements are in the same equivalence class or not. To *order* a list linearly, an ordering relation must be defined. Sorting can often be accomplished by means of ordering, when the ordering relation is so chosen that the identities of some type of characteristic represent the desired equivalence relation. As we shall see in the next section, searching is often carried out by sorting ordered lists.

† We use the term *codifying* here in preference to *coding* to distinguish the process being discussed in this chapter from the programming of a computer.

Generalized Concept of the Searching Problem in Terms of Boolean Algebra. The searching problem can be easily couched in terms of Boolean algebra. This is accomplished by observing that an item is uniquely defined (at least as far as the search is concerned) by the pattern of characteristics that are and are not associated with it. With three characteristics A_1, A_2, and A_3 a total of eight items X_0, X_1, X_2, X_3, X_4, X_5, X_6, and X_7 can be distinguished,

$$X_0 = \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 \qquad X_1 = A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 \qquad X_2 = \bar{A}_1 \cdot A_2 \cdot \bar{A}_3 \qquad X_3 = A_1 \cdot A_2 \cdot \bar{A}_3$$
$$X_4 = \bar{A}_1 \cdot \bar{A}_2 \cdot A_3 \qquad X_5 = A_1 \cdot \bar{A}_2 \cdot A_3 \qquad X_6 = \bar{A}_1 \cdot A_2 \cdot A_3 \qquad X_7 = A_1 \cdot A_2 \cdot A_3$$

Here, for example, $X_5 = A_1 \cdot \bar{A}_2 \cdot A_3$ means that item X_5 is associated with characteristic A_1 and A_3 but not with A_2. If the basic symbols are chosen to be the characteristics, then the columns of the basis can represent items distinguishable by these characteristics, where a unit in a row of a column indicates that the item of that column is associated with the characteristic of that row and a zero means that it is not. Thus

	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
$\#A_1 =$	0	1	0	1	0	1	0	1
$\#A_2 =$	0	0	1	1	0	0	1	1
$\#A_3 =$	0	0	0	0	1	1	1	1

The simple searching problem is then solved as follows: To find all items associated with $A_1 \cdot A_2$, simply form

				X_3				X_7
$\#A_1 \cdot A_2 =$	0	0	0	1	0	0	0	1

whence X_3 and X_7 are the desired items. Of course there is not always an item associated with every combination of characteristics; in such a case the columns for which there is no associated item are omitted from the basis, i.e., a *constrained* basis is used. Thus Table 11-1 can be considered as a constrained basis.

With this interpretation the searching problem can be generalized as follows: Suppose that it is desired to find all items associated with characteristic A_1 and at least one of the characteristics A_2 or A_3, that is, associated with $A_1 \cdot (A_2 + A_3)$. We simply form

$$\#A_1 \cdot (A_2 + A_3) = 0001\ 0101$$

whence the answer is X_3, X_5, and X_7. As another illustration, consider Table 11-1, and find all articles associated with nuclear theory *or* nuclear

plans *or both*. Observing that

Nuclear = *4.3* = 0000	1001	0011
Theory = *4.4* = 0010	0001	0011
Plans = *6.2* = 0100	1001	1110
we have $(4.3) \cdot [(4.4) + (6.2)]$ = 0000	1001	0011

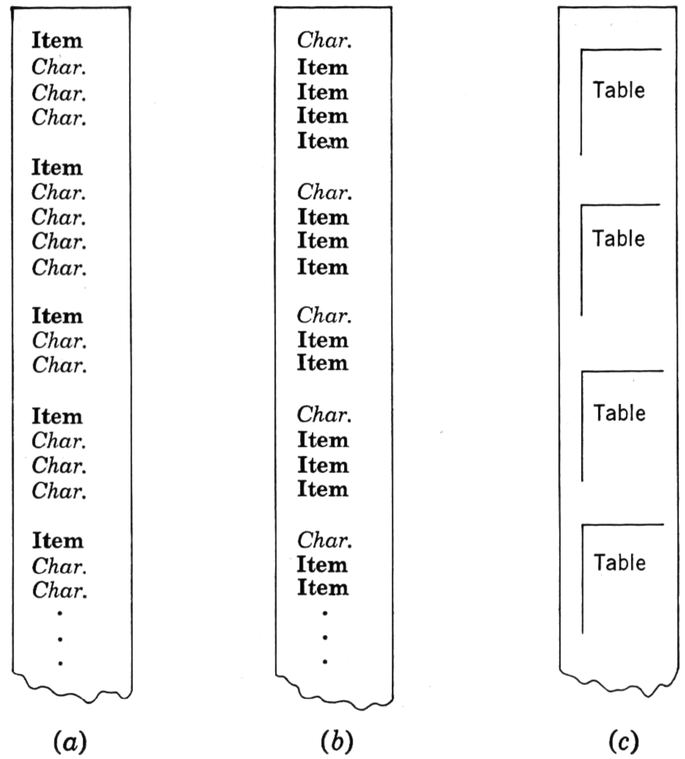
Hence articles **2.1**, **2.4**, **3.3**, and **3.4** by Nicholls, Seidle, Stockendal, and Tove are the final desired results. Summarizing, *when it is desired to find all items specified by a Boolean function of the characteristics, the designation number of the function is formed; the positions of the units indicate the desired items.*

11-2. Methods of Searching

The First Two Methods. For concreteness we assume in what follows that the items and characteristics are listed on magnetic tape. Following this assumption, we shall describe three methods for searching with a computer, using our illustrative bibliography as an example. Each of the searching techniques is, as will be shown, intimately associated with the listing arrangement of items and characteristics on the magnetic tape; this is because the *associations are indicated by means of the item-characteristic arrangement* on the tape.

First, consider the case where all characteristics associated with each item of the item list appear under that item (see Figs. 11-2*a* and 11-3*a*). Given a set of desired characteristics, the computer reads an item and its associated characteristics from the magnetic tape into the high-speed memory and compares to determine whether or not the given characteristics are included in the characteristics associated with the item; if so, the item would be suitably recorded. This process is accomplished for each item and its associated characteristics on the tape, one item at a time successively. In this way, after a single pass through the tape, every item associated with all the given characteristics is retrieved. The advantage of this system is that the items need not be ordered on the tape, since every item is searched. The disadvantage is that *every section* of the magnetic tape must be searched in detail, consuming much time in the computer.

Second, consider the case where the characteristic list is recorded on the tape and under each characteristic is a list of items associated with that characteristic (see Figs. 11-2*b* and 11-3*b*). The characteristics must appear in some order on the tape—e.g., alphabetically or numerically. Given a set of characteristics, the computer would first order them and then would go through the magnetic tape looking for the first, or smallest, characteristic; having found this, it would read the characteristic and its


Item
Char.
Char.
Char.
Item
Char.
Char.
Char.
Item
Char.
Char.
Item
Char.
Char.
Char.
Item
Char.
Char.
Char.
Item
Char.
Char.
Item
Char.
Item
Char.
Item

associated items into the high-speed memory and record all items associated with it. Continuing the search on the tape, the computer would look for the second, or next smallest, characteristic. Having found that, the computer would see which items associated with the second characteristic were *also* associated with the first and record them. Then the computer would look for the third characteristic and record only those items associated with it which were also associated with both the first two characteristics, and so on. The computer need not examine the whole tape in detail, but only those sections associated with each of the given characteristics; however, both the list of characteristics and the set of given characteristics must be preordered.

Note that, in terms of our association matrix of Table 11-1, the first and second methods described above correspond, respectively, to recording the successive *columns* of the matrix and the successive *rows* of the matrix on the magnetic tape.

The Tabledex Method. Finally, consider a third method for recording the items and characteristics on the magnetic tape. The method involves the recording of successive *tables* on the magnetic tape and is therefore called the *Tabledex* method. There is one table for each characteristic. A table is formed as follows: All items associated with this characteristic are listed in the table; adjacent to each item are *sublisted all those characteristics greater in order than the characteristic of the table* which are associated with that item (see Figs. 11-2*c* and 11-3*c*). For example, in Fig. 11-3*c*, consider the Tabledex table headed *4.4*. Observe from Table 11-1 that the characteristic *4.4* is associated with items **1.3**, **2.4**, **3.3**, and **3.4**. Now observe that item **2.4** is associated with *2.1*, *4.2*, *4.3*, *4.4*, *5.1*, *5.3*, and *6.2*, but *only* *5.1*, *5.3*, and *6.2* appear in the sublist, for only these are greater than *4.4*, the number of the table (see Fig. 11-3*c*). Similarly the sublists for **3.4**, **3.3**, and **1.3** were formed for this table. (Note that these sublists are ordered in the table on their first *characteristics*.)

The tables are listed in the order of their associated characteristic, and the items and the sublist of characteristics for each item are also listed in the proper order. Given a set of characteristics, the computer first puts them in the proper order, then searches the tape for the table associated with the lowest ordered characteristic, reading this into the high-speed memory, and finally searches the sublists, recording those items associated with a sublist containing all the other given characteristics For example, if the three given characteristics are *7.1*, *5.1*, and *5.2*, these would first be written in order as *5.1*, *5.2*, and *7.1*; then the table for *5.1* would be searched; and finally, if a sublist were found to contain both *5.2* and *7.1*, its item would be recorded. As can be seen in Fig. 11-3*c*, the items whose sublists contained *5.2* and *7.1* are **2.3**, **2.2**, and **3.2**; hence these are all items each of which is associated with all the given

characteristics. Using the Tabledex method, only the single section of the tape that contains the appropriate table need be searched in detail.

Population vs. Substantive Ordering of the Lists. The reader may have wondered how we arrived at the code number for the characteristics and items given in Fig. 11-1. The choice of numbering is related to the concepts of *population-based ordering* and *substantive-based ordering*. Substantive ordering refers to an ordering arrangement based on some attribute of the individual terms in the list. Thus an alphabetical ordering of items by author is a substantive-based ordering, an ordering of characteristics by research-field classification is substantive-based, etc. In Fig. 11-1 the *item*-list code numbers were assigned substantively to the articles, in alphabetical order; in addition the number of an item locates its column and line, e.g., item **2.3** is in column 2, line 3.

	1.1	
(*a*)	**3.4**	*3.1 3.2 4.1 4.2 4.3 4.4 5.1*
	7.1	
	1.2	
	1.4	
(*b*)	**2.1**	
	2.2	
	2.3	
	3.1	
	3.2	

FIG. 11-4. The first and last Tabledex tables for the bibliography of Table 11-1.

Population-based ordering is quite different: it refers to an ordering based on the *quantities of items or characteristics in the associations*. For example, population-based code numbers for each characteristic might be assigned as the number of items associated with it. To distinguish among characteristics that have the same number of associated items, a decimal number can be appended. The characteristics of our example are numbered in this way. Thus in Fig. 11-1 it is seen that *analysis*, *Netherlands*, *nuclear*, and *theory* each have four associated items; hence their population-based numbers are *4.1*, *4.2*, *4.3*, and *4.4*, respectively. Ordering the characteristics by population-based code numbers can save searching time. In the second method given above, it is most efficient to begin searching in the section of the smallest given characteristic when population-based numbers are used. (Why?) Similarly, in the Tabledex method when population-based numbers are used, tables that include *many* items have *short* sublists, and tables that have long sublists include *few* items. (Why?) For example, Fig. 11-4*a* illustrates the first and Fig. 11-4*b* the last Tabledex table for the bibliography of Table 11-1, for which the characteristics code numbers are population-based.

11-3. Manual Searching Methods

For each of the three searching methods given above there is an analogous manual searching method. Besides being of considerable importance in themselves, these manual methods can serve as a concrete visualization of the more abstract operations that take place in a computer.

Marginal-punch Cards. The first method given above corresponds to the use of marginal-punch cards, where each card represents an item and a marginal position on a card represents a characteristic. Different cards represent different items, but the same respective marginal position represents the same characteristic on each card. The association of characteristics with an item is made on its card by punching a notch in the marginal position of those characteristics which correspond to the particular item of the card. One type of card commonly used has holes corresponding to each marginal position. To sort a deck of the cards according to a particular characteristic, a needle is placed through the holes in the cards corresponding to this characteristic; then, when the needle is lifted, those cards which have been punched in this position will fall, the other cards will be lifted by the needle, and the deck is thereby separated into two parts (see Fig. 11-5). A search is performed by repeating this process successively on the dropped cards (which have the desired characteristic) until those cards punched in at least all the desired characteristic positions are obtained. Note that the initial deck of cards need not be kept in any particular order whatsoever.

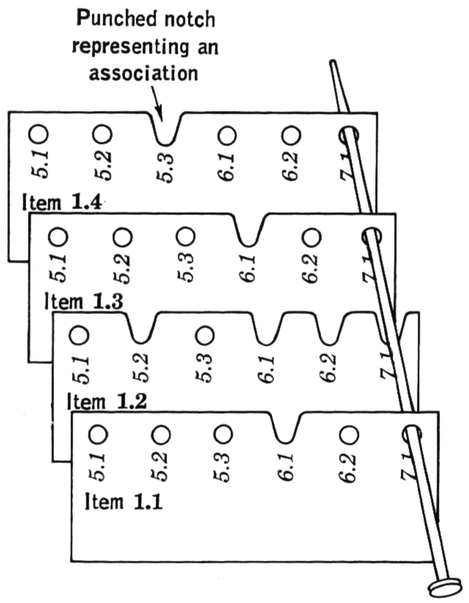


FIG. 11-5. Marginal-punch cards.

Peek-a-boo Cards. The second method given above corresponds to the use of the so-called "Peek-a-boo cards." Here each card corresponds to a characteristic; the surface of a card is laid off in an XY coordinate system with each particular pair of coordinates representing an item. Different cards represent different characteristics, but the same respective XY coordinates represent the same item on each card. The association of characteristics with items is made by punching a hole at the XY coordinate positions that represent each item associated with the characteristic of the card. A search is made by first selecting from a deck of cards those cards which correspond to the given desired characteristics. The selected cards are placed together to form a pack; if an item is associated with all the given characteristics, then there will be a hole passing completely through the pack at the coordinate point corresponding to that item (see Fig. 11-6). Note that the initial deck of cards must be kept in some definite order so that the cards can be easily selected for a particular search, and note that the pack of selected cards must be carefully aligned.

Programming a Computer to Generate a Manual Tabledex Collection. The third method given above corresponds to the manual use of the Tabledex tables themselves. Here the tables are printed in conventional

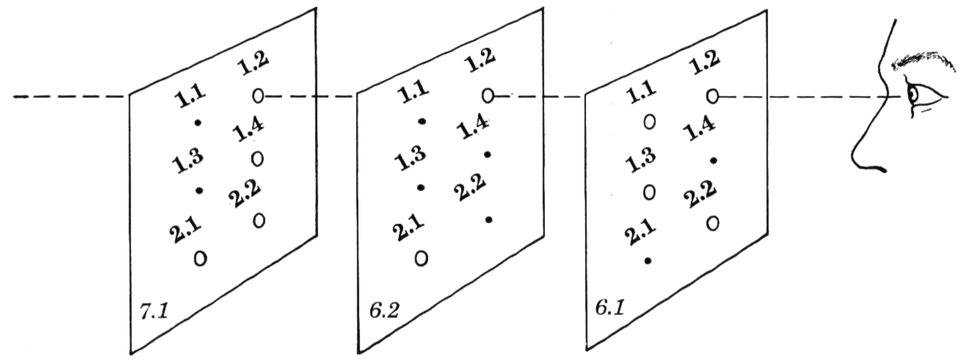


FIG. 11-6. Peek-a-boo cards.

bound-book form. The use of a collection of tables is directly analogous to their use on magnetic tapes. For example, suppose that we have a collection of tables corresponding to our bibliography illustrated above and that we desire to find all articles on the *application* of *nuclear theory*. From the alphabetical word list the code numbers are found: *application* *5.1*, *nuclear* *4.3*, *theory* *4.4*. These are placed in numerical order: *4.3*, *4.4*, and *5.1*. One then turns in the collection of tables to Table *4.3*, which would appear as shown:

TABLE *4.3*

✓	**2.4**	*4.4*	*5.1*	*5.3*	*6.2*
✓	**3.4**	*4.4*	*5.1*		
	3.3	*4.4*	*5.3*	*6.1*	*6.2*
	2.1	*6.2*	*7.1*		

The first two rows of Table *4.3* contain both *4.4* and *5.1* and are checked. Hence articles **2.4** and **3.4** by Seidle and by Tove are associated with the given words. The advantage of the manual use of Tabledex lies in its

being in bound-book form, without the necessity for the manipulation of cards.

A computer can be programmed to automatically compile and print a Tabledex collection for manual use. The method used to compile the Tabledex collection is of particular interest as an application of the techniques of this chapter. First the characteristics of each article-characteristic association are ordered in themselves. For example, we would have the article-characteristic association: item **2.4**, characteristics *2.1*, *4.2*, *4.3*, *4.4*, *5.1*, *5.3*, and *6.2* (see Table 11-1). Second, from each such article-characteristic association, a list of all possible rows of the tables can be generated. For example, from the associations just illustrated the following seven rows can be generated:

2.4	*2.1*	*4.2*	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*
2.4	*4.2*	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*	
2.4	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*		
2.4	*4.4*	*5.1*	*5.3*	*6.2*			
2.4	*5.1*	*5.3*	*6.2*				
2.4	*5.3*	*6.2*					
2.4	*6.2*						

Third, the list of all such rows generated from every article-characteristic association is then sorted by ordering on the *first* (*leftmost*) *characteristic code number*. After the ordering, all rows with the same first characteristic appear adjacent. For example, the following rows appear adjacent:

2.4	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*
3.4	*4.3*	*4.4*	*5.1*		
3.3	*4.3*	*4.4*	*5.3*	*6.1*	*6.2*
2.1	*4.3*	*6.2*	*7.1*		

When the number *4.3* has been removed, the adjacent rows are just Table *4.3*. Thus from this ordered list of rows the tables can be printed out by the computer.

11-4. Searching with Relaxed Conditions

The Problem of Relaxed Conditions. It frequently happens that no item is associated with all the characteristics given. In such a case it is usually advisable to *search with relaxed conditions.*

For example, suppose that there are given 10 equally important characteristics and that no item is associated with all of them. Then the natural question to ask is: Are any items associated with all but one of the characteristics, i.e., with any 9 of the characteristics? If not, then are there any items associated with all except 2 of the characteristics, i.e., with any 8 of the 10, and so forth? *We shall describe a procedure for accomplishing this important kind of search which does not involve trying all possible combinations of* 9 *of the* 10 *characteristics, then all possible combinations of* 8 *of the* 10, *etc.* This procedure can be extended to include a second type of relaxed-condition problem which may arise, as follows: Suppose that 3 of the 10 given characteristics are most important and that it is desired to find all items associated with these 3 characteristics and also any 6, or any 5, etc., of the remaining 7 characteristics. The procedure can be easily generalized to include almost all relaxed-condition situations that occur in searching.

The idea of a *simple sort* can be used to compare searching techniques. A simple sort is a separation of a collection of items into two parts by means of a *single* characteristic C, that is, into a part whose items are each associated with C and a part whose items are not. A simple sort is analogous to the manual separation of a deck of marginal-punch cards into two parts by means of a single needle.

To see the advantage of the searching technique to be described over the straightforward method of trying all possible combinations of leaving out one of the characteristics at a time, then two at a time, then three at a time, . . . , then $K = n - r$ at a time, note that the total number of simple sorts required for the straightforward method is

$$\sum_{k=0}^{K} \frac{n!}{(n-k-1)!}$$

However, the number of simple sorts required by the method to be presented is simply

$$\frac{(K+1)(n-K)}{2}$$

Searching for Items Associated with Any r Out of n Characteristics. The method of procedure is illustrated by Fig. 11-7. Suppose that there are four given characteristics, represented by C_1, C_2, C_3, and C_4. We would first sort the collection of items for those associated with C_1. Next we would sort those which *have* C_1 for those which also have C_2. Then those having *both* C_1 and C_2 would be sorted for those also having C_3, and so forth, until we find all items that have C_1, C_2, C_3, and C_4. In Fig. 11-7 we indicate the collection of items that are associated with *at least* C_1 by C_1, those associated with at least C_1 *and* C_2 by $C_1 \cdot C_2$, etc. The collection of items that do *not* have C_1 is indicated by $\bar{C}_1$, etc. Then $C_1 \cdot C_2 \cdot \bar{C}_3$ would represent the collection of items that at least have C_1 and C_2 and not C_3. Thus the results of each successive sort of the *first pass* through the collection of items are indicated by the first row of Fig. 11-7. Here, in the first sort, the collection of items is divided into two parts, the part C_1 and the part $\bar{C}_1$. Then the collection with C_1 is sorted for C_2, resulting in a part $C_1 \cdot C_2$ and a part $C_1 \cdot \bar{C}_2$, and so forth. After the first pass there will result five collections $\bar{C}_1$, $C_1 \cdot \bar{C}_2$, $C_1 \cdot C_2 \cdot \bar{C}_3$,

$C_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$, and $C_1 \cdot C_2 \cdot C_3 \cdot C_4$; all but the last of these will be used if another pass is necessary.

If the collection $C_1 \cdot C_2 \cdot C_3 \cdot C_4$ is empty, i.e., if no items are associated with all of $C_1 \cdot C_2 \cdot C_3 \cdot C_4$, we proceed to the second pass (the second row of Fig. 11-7). This time we start by first sorting $\bar{C}_1$ for C_2, obtaining $\bar{C}_1 \cdot C_2$ and $\bar{C}_1 \cdot \bar{C}_2$. We put the collections $\bar{C}_1 \cdot C_2$ and $C_1 \cdot \bar{C}_2$ together; these are sorted for C_3, and two collections are obtained, one with $\bar{C}_1 \cdot C_2 \cdot \bar{C}_3$ and $C_1 \cdot \bar{C}_2 \cdot \bar{C}_3$, the other with $\bar{C}_1 \cdot C_2 \cdot C_3$ and $C_1 \cdot \bar{C}_2 \cdot C_3$. This latter collection is put with $C_1 \cdot C_2 \cdot \bar{C}_3$ and these sorted for C_4 to obtain two collections, one with $\bar{C}_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$, $C_1 \cdot \bar{C}_2 \cdot C_3 \cdot \bar{C}_4$, and $C_1 \cdot C_2 \cdot \bar{C}_3 \cdot \bar{C}_4$, the other with $\bar{C}_1 \cdot C_2 \cdot C_3 \cdot C_4$, $C_1 \cdot \bar{C}_2 \cdot C_3 \cdot C_4$, and $C_1 \cdot C_2 \cdot \bar{C}_3 \cdot C_4$. This latter collection and collection $C_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$ form the collection of all items having all except one of the given characteristics. If this collection is empty, then we proceed to a third pass, as shown in the third row of Fig. 11-7, to find the collection of items with all except two of the characteristics, and so forth.

Searching with Permanent Conditions. Suppose that there were seven characteristics given, C_1, C_2, . . . , C_7, and that no item of the collection under consideration were associated with all these seven characteristics. The next step would be to relax the conditions, but suppose that we must have items associated with C_5, C_6, and C_7, and with as many of the others as possible. The characteristic combination $C_5 \cdot C_6 \cdot C_7$ is called a *permanent condition.* The searching technique would be first to sort the collection to obtain all items associated with $C_5 \cdot C_6 \cdot C_7$. Then we would sort these on C_1, C_2, C_3, and C_4, as described above, to obtain all items each associated with $C_5 \cdot C_6 \cdot C_7$ *and* three, or two, or one of C_1, C_2, C_3, C_4.

As another illustration of a permanent condition, suppose that we desired to obtain all items associated *with one or more* of C_3, C_6, and C_7, and with as many of C_1, C_2, C_3, and C_4 as possible. In such a case we would first sort for C_5, sort the remaining items for C_6, and then sort those still remaining for C_7. These three sets would be put together to form the collection of items having one or more of C_5, C_6, and C_7. We would next sort this collection on C_1, C_2, C_3, and C_4, as described above, to obtain all items each of which is associated with C_5, C_6, and/or C_7, and three, or two, or one of C_1, C_2, C_3, and C_4. In either of these examples, if no items were to have the permanent conditions, no further sorting could be accomplished.

Searching with Preferential Conditions. Suppose that there were nine characteristics given, C_1, C_2, . . . , C_9, and that no item of the collection were associated with all nine given characteristics. However, suppose that a preference were associated with the characteristics so that, say, C_1, C_2, and C_3 have the most preference, C_4, C_5, and C_6 have the next preference, and C_7, C_8, and C_9 have the least preference. The search-

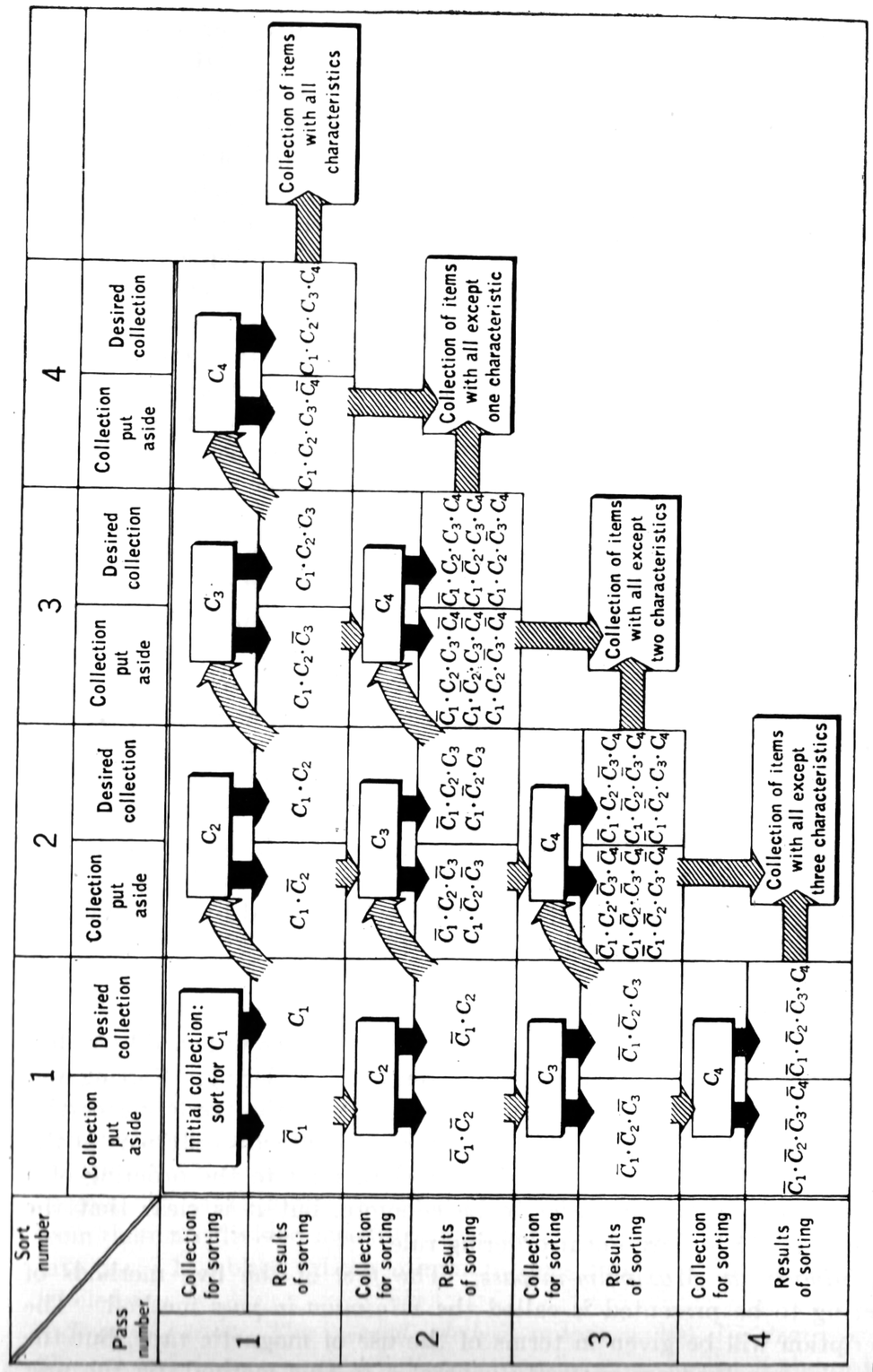


FIG. 11-7. Analysis of method of searching for all items each of which is associated with all, all except one, all except two, etc., of the given characteristics C_1, C_2, C_3, and C_4.

ing procedure would first try for all of C_1, C_2, and C_3, or at least two of C_1, C_2, and C_3, or at least one, etc., as described above. Having chosen the items associated with the greatest number of the most preferred characteristics, these would be considered a permanent condition and would be sorted for all of C_4, C_5, and C_6, or at least two of C_4, C_5, and C_6 or at least one, etc. From the items having the most preferred characteristics we would now have chosen the items with the most characteristics of the next preference. Considering these as representing a permanent condition, we would try next for the items with the most characteristics of the least preference, that is, C_7, C_8, and C_9.

Note that the process of this section would be the natural one to follow in the problems of the previous paragraph when no items satisfy the desired permanent condition. The initial permanent condition would then be stated in terms of preferential conditions, and the process of this section would be used to obtain an approximation to the initial permanent condition.

Relaxing Sets of Conditions. Suppose that it is desired to search for all items associated with at least one of C_1, C_2, and C_3, *and* with at least one of C_4, C_5, and C_6, *and* with at least one of C_7, C_8, and C_9, *and* with at least one of C_{10}, C_{11}, C_{12}. Suppose that no such item exists; then the problem is to find all items that have at least one characteristic from at least three of the four sets of characteristics, or from at least two of the four sets, and so forth. Here we wish to relax sets of conditions. In order to facilitate the discussion, let us represent condition C_1, C_2, and/or C_3 by C^1, condition C_4, C_5, and/or C_6 by C^2, condition C_7, C_8, and/or C_9 by C^3, and condition C_{10}, C_{11}, and/or C_{12} by C^4. Then the searching technique becomes as described and illustrated above, with superscripts replacing subscripts, and with the single sorts replaced by the combinations of sorts necessary to determine the C^i.

11-5. Sorting and Ordering

As discussed in Sec. 11-1, sorting can always be accomplished by ordering; in this section we shall turn our attention to ordering. Sorting and ordering are essentially the same process, except that ordering carries the process all the way, while sorting carries the process only part of the way. In our discussion of ordering we shall refer to the ordering of a given list of *numbers* in order to be concrete, but it is clear that the methods apply for any linear ordering rule.

Minimum- or Maximum-in-pass. The *first* of the five methods of ordering to be presented is called the *minimum-in-pass* method. The description will be given in terms of the use of magnetic tape, but the principles involved are perfectly general. In this method, on the first pass over the tape the minimum number is recorded as follows: The first number is recorded in a temporary cell and is compared successively with the other numbers until a smaller number appears. The first number replaces this smaller number on the tape as this number is placed in the temporary cell. The successive comparisons are continued until another smaller number appears; again the two numbers are interchanged, and so forth. At the end of the first pass it is clear that the smallest number appears in the temporary cell. This is placed as the first number on the tape, and the *second pass* is started with the *second number* on the tape, proceeding as before. At the end of this pass the second smallest number appears in the temporary cell and is placed as the second number on the tape, and so forth. The ith pass starts with the ith item, determines the ith smallest number, and puts it in the ith position on the tape. For N numbers this method requires $N - 1$ passes through the tape; however, if the tape can be positioned, starting each pass with the next number in turn effectively reduces the number of passes by half, to approximately $N/2$.

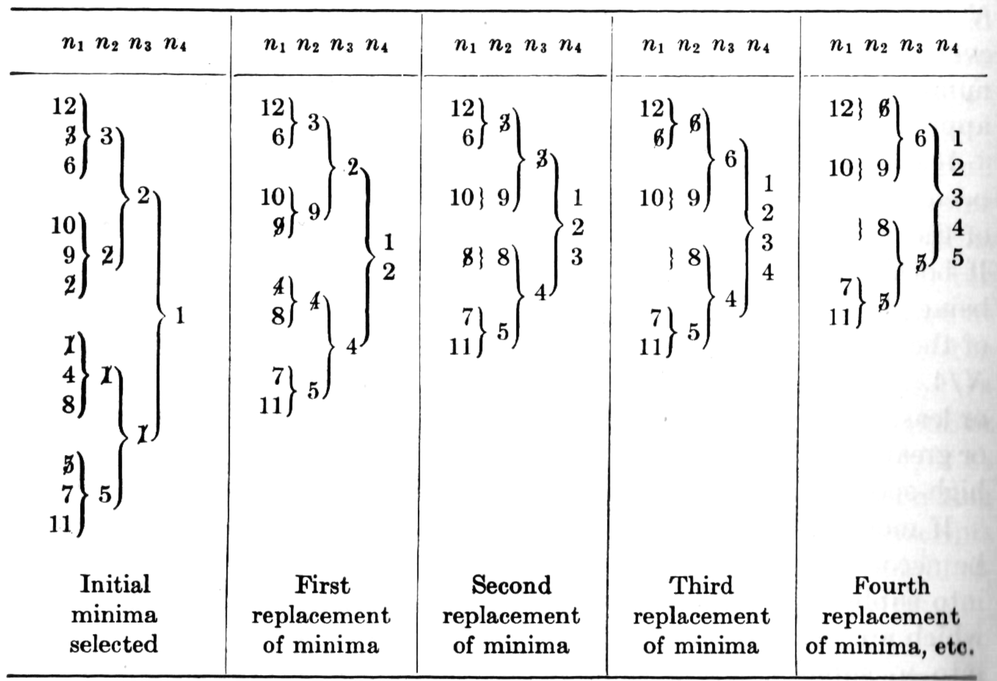
It is obvious that the maximum instead of minimum number could have been recorded on each pass (*maximum-in-pass*) and placed at the end of the remaining list; this method would again require about $N/2$ passes. If both the minimum and maximum are recorded in each pass, the latter being placed on the tape at the end of the pass, the former at the beginning of the next pass, the number of passes is again halved, to approximately $N/4$. Depending on the size of the high-speed memory, the least two, or least three, or least m numbers, and the greatest two, or greatest three, or greatest n numbers could be found on each pass (and ordered within the high-speed memory), further reducing the number of passes required.

If more than one tape is available, the minimum-in-pass method can be accomplished in parallel. The original list to be ordered is divided into sublists (say n_1 in number). A new list is made from the minima which are *removed* from each of the n_1 sublists. This new list is divided into n_2 sublists, and a third list is made from the minima which are *removed* from each of the n_2 sublists. As many new lists as desired can be made, and the first number of the ordered list will be the minimum of the last such list. As a number is removed from a list, it is replaced by the minimum from the sublist from which it was originally taken (until the sublist is exhausted). Table 11-2 illustrates the process applied to an original list of 12 numbers, divided into $n_1 = 4$ sublists. The minima from these are divided into $n_2 = 2$ sublists; their minima are then divided into $n_3 = 1$ sublist, whose minimum goes into the final ordered list, labeled n_4.

Transposition. The *second method* orders a list by *transposition.* The process is to transpose adjacent numbers on the list if the order of their characteristics is inverted. It starts by comparing the first two

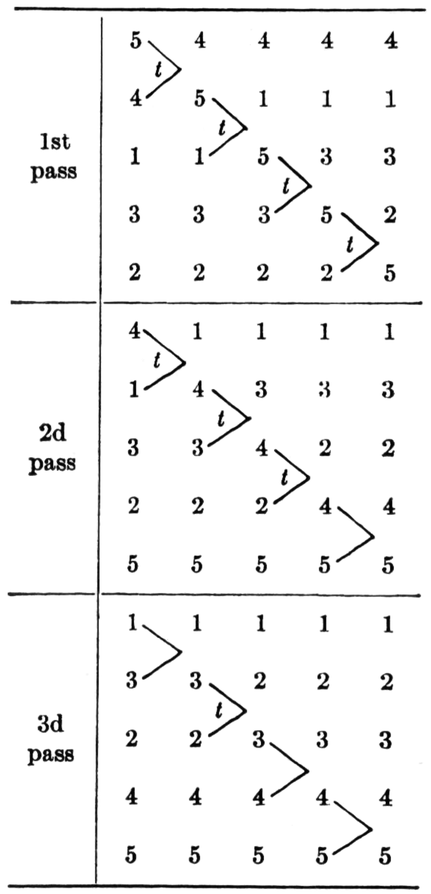
numbers and continues on *overlapping pairs* through the list. Pass after pass is made until no transpositions are required on an entire pass; then the list is in order. For example, consider Table 11-3 (where the caret represents comparison, the t transposition). Here three passes were necessary to order the numbers. The fourth pass will result in *no* further transpositions, indicating that the list is in order. A variation on this method is to order successive overlapping sets of three or more (say r) numbers; a single tape may still be used in this process, and the number of passes required is approximately divided by $r - 1$. Table 11-4 illustrates the procedure for overlapping sets of three numbers.

TABLE 11-2. MINIMUM-IN-PASS ORDERING USING MULTIPLE SUBLISTS



Ordering by Radix Coefficients. The *third method* is probably the most obvious of all the methods when a clearly defined radix occurs; in the present discussion we shall consider decimal numbers (i.e., radix ten). The method consists in sorting the numbers by the most significant figure, putting the sorts in order. The numbers within each sort are ordered on the second most significant figure; then *within each of these subsorts* the numbers are ordered on the third most significant figure; and so forth. For an example, see Table 11-5. This method has the disadvantage that after the first pass the computer must work on subregions of the tape, which can present difficulties. For sorting punch cards, for example, this means that individual sorts must be made on successively smaller

TABLE 11-3. ORDERING BY TRANSPOSITION†



† Successive overlapping pairs are scanned.

TABLE 11-4. ORDERING BY TRANSPOSITION WITHIN OVERLAPPING SETS OF THREE NUMBERS



TABLE 11-5. ORDERING BY THE THIRD METHOD

Original list	Sort by most significant figure	Sort by 2d most significant figure	Sort by 3d most significant figure
345	256	256	255
344	265	255	256
537	255	265	264
256	264	264	265
554	345	337	337
447	344	345	344
337	337	344	345
265	347	347	347
255	447	447	446
466	466	446	447
446	446	466	464
464	464	464	466
347	537	537	336
264	554	536	337
536	536	544	544
544	544	554	554

and smaller pieces of the original deck; such handling becomes increasingly inconvenient.

The *fourth method*, which is essentially the inverse of the third, eliminates this difficulty. Here the first sort is made on the least significant figure. Then the entire tape is sorted on the second least significant figure, and so forth. For an example, see Table 11-6. However, both these methods offer the difficulty that simultaneously used multiple tape units are indicated. Of course the subsorts might be made by one of the first two methods, whence the entire sorting procedure becomes a mixture of methods.

Suppose that for the fourth method ten bins, or magnetic-tape units, in addition to the unit holding the original list, were not available for the separation by least significant decimal digit. For example, suppose that only 5 units were available, 1 of which contains the original list. Then we could convert the least significant decimal digit of each number to two radix-4 digits and in two passes order the list by the least significant decimal digit. Similarly two passes would be required for the next least significant decimal digit, and so forth.

TABLE 11-6. ORDERING BY THE FOURTH METHOD

Original list	Sort by least significant figure	Sort by 2d least significant figure	Sort by 3d least significant figure
345	344	536	255
344	554	537	256
537	464	337	264
256	264	344	265
554	544	544	337
447	345	345	344
337	265	446	345
265	255	447	347
255	256	347	446
466	466	554	447
446	446	255	464
464	536	256	466
347	537	464	536
264	447	264	537
536	337	265	544
544	347	466	554

11-6. Ordering by Merging

Merging. The *fifth method* is based on the simple idea of the *merging of two ordered lists.* For example, consider these two ordered lists of numbers:

265		255
337		264
446	and	464
466		536
		544

Forming a single ordered list containing all the numbers is accomplished by successively comparing the first numbers on each list and removing the smaller number to a third list. Thus, since $255 < 265$, 255 becomes the first number of the new list; now 264 becomes the first number of the

second list, and since 264 < 265, 264 becomes the second number of the new list; next, since 265 < 464, 265 becomes the third number of the new list; and so forth. In this manner the two ordered lists are *merged* to form a single new ordered list, namely,

255
264
265
337
446
464
466
536
544

In more mathematical notation, if the two ordered lists are $a_1 < a_2 < \cdots < a_I$ and $b_1 < b_2 < \cdots < b_J$, then the merged list, namely $C_1 < C_2 < \cdots < C_{(I+J)}$, is formed, as indicated in the flow diagram of Fig. 11-8.

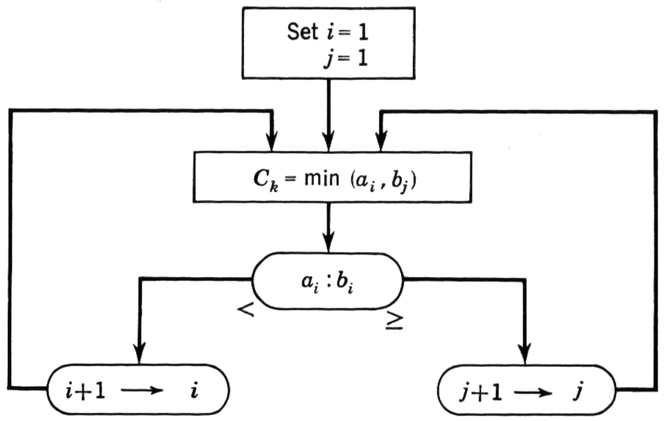


FIG. 11-8. Flow chart for merging two ordered lists.

The initial step of the general ordering by merging process is to separate the given list into *two* lists in a manner that preserves any ordering that might have originally appeared among adjacent numbers. The first number of the given list is used to start the first of two generated lists. Then successive numbers of the given list are added to this new list so long as they appear in the proper order. The first number encountered that appears out of order is used to start the second generated list All successive numbers that are in the proper order are placed successively in this second list. When a number out of order again appears, it is placed in the *first* list, with those following numbers which are in the proper order, and so forth. Thus in forming the two new lists we switch output

lists whenever a number appears out of order, so that any order within a string of adjacent numbers in the given list will be retained in one of the two lists. For the example below, strings of numbers in the original list that appear in the proper order are included in braces; these are alternately placed in the two generated lists.

Original list	*New list*	*New list*
{345	{345	{344
{344	{256	{537
{537	{554	{447
{256	{337	{265
{554	{255	{446
{447	{466	{464
{337	{347	{264
{265		{536
{255		{544
{466		
{446		
{464		
{347		
{264		
{536		
{544		

When this initial step is finished, the ordering proceeds as follows: Each of the two lists contains sublists that are ordered, and each sublist of one can be paired with a sublist of the other, except perhaps the last sublist of one of the lists (see the horizontal lines below). The pairs of ordered sublists are then merged. The merged results are again placed alternately† in two new lists as shown below:

First merge results			*New list*	*New list*
345	merge	344	344	256
		537	345	447
256	merge	447	537	554
554			265	255
337	merge	265	337	264
		446	446	466
		464	464	536
255	merge	264		544
466		536	347	
		544		
347				

† The alternation of the merged sublists in the new lists ensures the maximum number of pairings of sublists for the next merge; if the total number of sublists is known, the same result would be obtained by putting the first half on one tape, the second half on the other.

The process is repeated, forming each time two lists of increasingly longer ordered sublists, until all the numbers appear in one ordered list:

Second merge results		*Third merge results*		*Ordered list*
256	255	255	347	255
344	264	256		256
345	265	264		264
447	337	265		265
537	446	337		337
554	464	344		344
	466	345		345
	536	446		347
	544	447		446
347		464		447
		466		464
		536		466
		537		536
		544		537
		554		544
				554

Often four magnetic tape units are used in this ordering process. Two of the tape units hold the old lists, and the other two record the

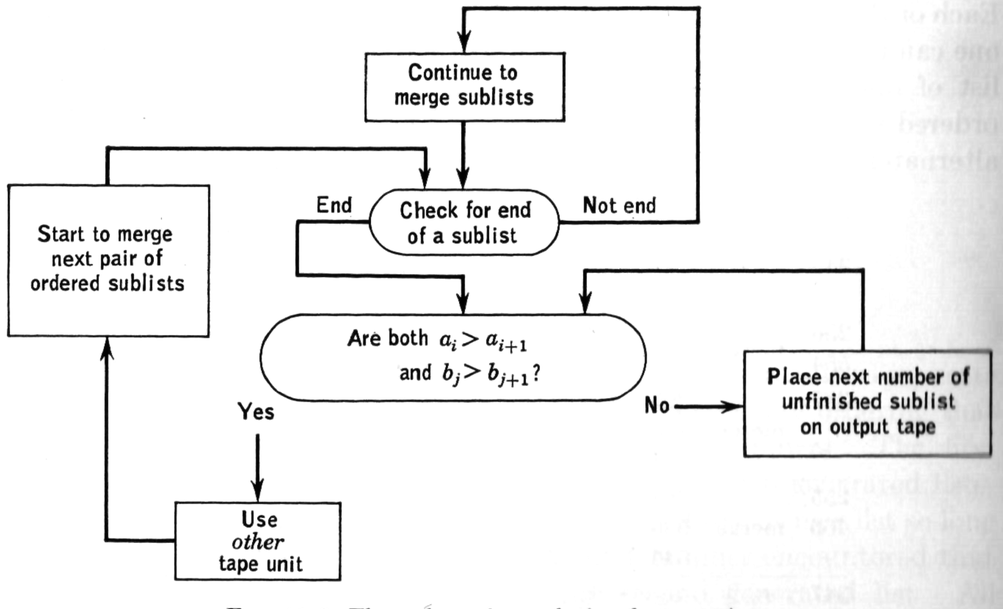


FIG. 11-9. Flow chart for ordering by merging.

results of the merging. Then, when both old lists have been exhausted, the newly formed lists become the old lists and the pairs of tape units

exchange functions. In merging two sublists, the end of a sublist is sensed when $a_i > a_{i+1}$; when both sublists are exhausted, i.e., when both $a_i > a_{i+1}$ and $b_j > b_{j+1}$, then the merge starts on the next pair of sublists, the results being put on the *other* receiving tape unit. Figure 11-9 gives a flow chart for this process.

Variations on Merging. The technique of ordering by merging two ordered lists is easily extended to ordering by merging more than two ordered lists. The initial separation of a list to be ordered can involve more than two new lists, and then the merging of ordered sublists would be accomplished successively with one sublist from each. If the initial step formed m new lists and the merging of sublists from these formed n new lists, then $m + n$ magnetic tapes would be involved. Below we illustrate the case for three initial new lists (i.e., $m = n = 3$):

	Initial step						*Ordered list, second merge results*
Original list	*New list*	*New list*	*New list*	*First merge results*			
345	345	344	256	256	255	264	255
344	447	537	554	344	265	536	256
537	255	337	265	345	337	544	264
256	466	446	347	447	347		265
554		464		537	446		337
447	264			554	464		344
337	536				466		345
265	544						347
255							446
466							447
446							464
464							466
347							536
264							537
536							544
544							554

Another variation takes further advantage of any intrinsic ordering in the original list. Here a tape is not closed off as a smaller item occurs, but rather is held open to receive any larger item later in the list. Each new item is compared with the final items on each tape, in order of largest to smallest, and is put on that tape whose last item is as large as possible without exceeding the new item. If the new item is smaller than all the final items, it is placed on the tape with the largest item to start a new sublist. Using this technique to sort the above list of numbers, merging from two lists, gives the following (with the sublists indicated):

Initial step		*First merge results*		*Second merge results*	
345	344	344	256	256	255
537	*554*	345	337	337	264
256	337	537	447	344	265
447	255	554	*464*	345	347
265	446	255	*466*	447	446
466	464	265	264	464	536
347	264	446	347	466	544
536		536		537	
544		544		554	

The third merge, instead of the fourth required above for two-tape merging, will give the ordered list. Note how changing tapes for the italicized numbers allows the lengthening of the sublists.

Comparison of Ordering Methods. The number of passes, i.e., the number of times one must look through all the numbers of the original list, is very large for the first two methods as compared with the merge method. The straight minimum-in-pass method requires $N/2$ (or $N/4$ or perhaps less depending on the equipment) passes for N numbers. The merge sort using $2T$ magnetic tape units requires no more than $K + 1$ passes, where K is the smallest integer such that $T^K \geq N$. The merge sort has the additional feature that full advantage is taken of any ordered sublists that may appear in the original list, and if the numbers are randomly distributed, there ought to be about $N/2$ ordered sublists. In such a case $1 + K'$ passes are required, where K' is the smallest integer such that $T^{K'} \geq N/2$. For the transposition method there can be as many as $n(n + 1)/2$ passes, but here again any ordered sublists that may appear in the original list can reduce this number of passes. The variation of the transposition method is of course more efficient. Ordering by the radix-coefficient methods can compete with the merge sort when all numbers from 1 to N are in the list, with practically no ordered sublists. But if the numbers have more than K digits, where K is the smallest integer such that $R^K > N$, where R is the radix being used, then the merge sort would require fewer passes. Since the radix-coefficient methods do not take advantage of any ordering that might already exist in the list, the merge sort can be still more advantageous. However, if only a single tape is available, only the minimum-in-pass or transposition can be done.

Of course the number of passes required for ordering a list is not necessarily the only criterion applied in deciding on a method for ordering. The nature of the equipment available, e.g., how many magnetic-tape units, or bins, are available and whether these can or cannot be used concurrently, how complex operations can be performed, etc., often limits the types of methods that can be used. The number of items or numbers

to be sorted, the number of times such a process must be repeated, etc., all enter into the choice of the ordering system. In general a combination of ordering techniques can be used to take advantage of particular situations that may occur at the different levels of the ordering process. However, if we were asked to choose which system is of greatest importance, we would say the merge ordering method and its variations have the most advantages.

11-7. Codifying: Error Correction and Superimposition

Definitions. By *codifying*† is meant the process of relating to an item or characteristic a *set of symbols*, called a *codification*. As we have seen (cf. Sec. 11-1), codifying is intimately associated with searching, sorting, and ordering. Codifying is used for three purposes: first just for convenience in handling the information in a shorter form rather than in raw form; second for making the information compatible with a particular mechanical or electronic processing device being used; and third to facilitate the particular method of searching, sorting, or ordering being used.

In general, codifying involves two processes, called *encoding* and *decoding*. Encoding is the process of generating the codification related to a given item or characteristic; decoding, conversely, is determining the items or characteristics related to a given codification. The processes of encoding and decoding are accomplished by means of either a *code book* or a *relation scheme*. For example, a telephone book is an encoding code book that gives the codification (i.e., the telephone number) related to an item (i.e., a person); the decoding is accomplished after dialing, by means of a scheme built into the telephone company's electrical dialing equipment. Another example is the biological codification of the genes, where the encoding is the evolutionary process and the decoding is the ontogenetic process that results in, say, an individual animal.

The codification of items or characteristics can be either *substantive* or *nonsubstantive* in character. A substantive codification is one in which the symbols have meaning in themselves. A substantive codification of names, for example, might comprise simply the first four consonants of the last names with the initials. A word in a dictionary is a substantive codification of its meaning. The genes, of course, represent an outstanding example of substantive codification.

For a nonsubstantive codification consider the example of the codification of research reports by accession numbers, or the codifying of an item in a bibliography by the page and line number on which it is found. Nonsubstantive codifications are frequently merely *addresses* that tell the locations of the related items or characteristics. An exception to this is the population-based (nonsubstantive) codification considered in

† See footnote on p. 498.

Sec. 11-2, in which the symbol for a characteristic is the number of items associated with that characteristic.

The relationships between encoding, decoding, and information processing are summarized in Fig. 11-10. For example, consider our illustrative bibliography discussed above. Here the encoding was the relating of population-based numbers to the given characteristics (i.e., the given retrieval words) by means of a code book. The processing was searching, where, for instance, the Tabledex method might be used, which resulted in a collection of numbers related to the associated items (i.e., the page and line numbers of desired article references). Finally the decoding process was accomplished by locating the items at their address (i.e., the entries are retrieved from their page and line numbers). The information processing here consisted in transforming a set of characteristic codifications into a set of item codifications.

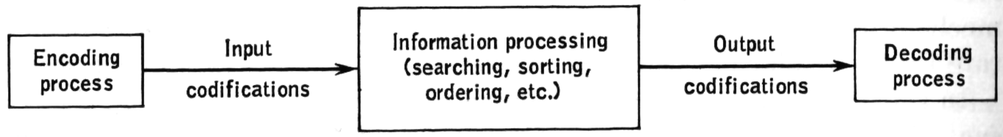


FIG. 11-10. Relationships between encoding, decoding, and information processing.

Efficiency of Codifications. To determine how many symbol *positions* are required in an unambiguous codification using K different symbols, *we consider each symbol as a number in the base (radix) K and use enough symbol positions to count the number of items in the base K.* That is, with N items (or characteristics) and K symbols, *the number of symbol positions in a codification must be no smaller than the smallest integer n such that $K^n \geq N$ (or $n \geq \log_K N$) if the codification relating to an item is to be unique in the list.* For example, for 600 items and a binary codification (that is, $K = 2$), we have†

$$n \geq \log_2 600 = \frac{2.7782}{0.3} = 9.261$$

or $n = 10$. If the number of positions is greater than n, that is, if more positions are being used than are necessary, the codification is called *redundant.* If fewer than the required n positions are used, the codification is called *irredundant* and the code for an item is *not* unique in the list. A codification that is neither redundant nor irredundant is called *efficient.*

For example, an English word can be considered as a codification of an idea using the 26 alphabetical symbols. Suppose that we have a

† Recall that $\log_K N = \log_{10} N/\log_{10} K$. For $K = 2$, $\log_2 N = \log_{10} N/\log_{10} 2$, where $\log_{10} 2 \approx 0.3$.

dictionary all the words of which are no longer than 10 letters. We can make all the words the same length by introducing a 27th letter, say α, and filling up with α's all positions to the left of a word to make 10 positions altogether, if necessary. It is then possible to have more than 145×10^{12} different words in our dictionary. However, a desk dictionary has only approximately 100,000, or 10^5, words, and so a dictionary is redundant. Then again, that's to be expected.

In the next paragraphs we shall consider a use for purposely redundant codification; that discussion will be followed by consideration of a use for purposely irredundant codifications.

Self-correcting Redundant Codification. Frequently situations arise where errors may be introduced into a codification during storage or transmission. If errors cannot be tolerated, a redundant code can be devised with the extra positions of the redundancy used to correct such

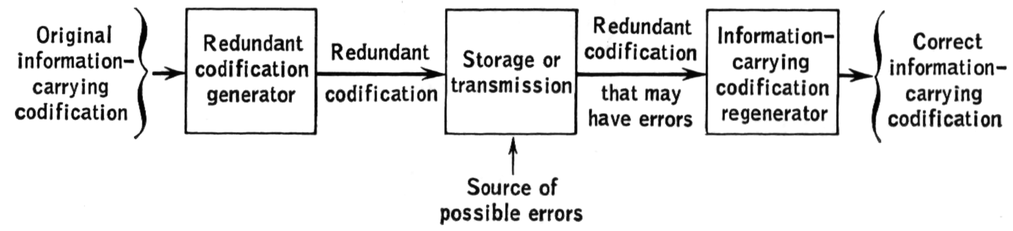


FIG. 11-11. The self-correcting redundant-codification process.

errors as may occur. However, the self-correcting scheme that makes use of redundant positions *must also be able to correct errors* that may occur in these extra *redundant positions* as well. The process can be outlined thus: First from the given information-carrying codification there is generated the redundant codification. This redundant codification is stored or transmitted, etc., during which time some errors can enter into the codification. Finally, when the information is to be used, there is generated from the redundant codification (which may contain some errors) the *correct* information-carrying codification in spite of any errors in the redundant codification (see Fig. 11-11). In what follows we shall confine ourselves to *binary codification.*

Let us first illustrate a self-correcting technique that will correct a single error. The first step is the formation of the redundant codification. Suppose that the information-carrying part of the codification has 4 bits, I_0, I_1, I_2, I_3—say, for example, 1101. We form an array as shown in Fig. 11-12 (with the top row all units and the columns otherwise having all possible combinations of zeros and units). The information-carrying codification is set down beside the array as a column. The desired redundant codification is formed bit by bit by comparing this

column successively with each column of the array, counting the number of units in common (i.e., logically multiplying the columns and counting the number of units in common). If the number of units is even, a zero

		0123	4567
Information-carrying codification	$I_0 = 1$	1111	1111
	$I_1 = 1$	0101	0101
	$I_2 = 0$	0011	0011
	$I_3 = 1$	0000	1111
Redundant codification		1010	0101

FIG. 11-12. Work for forming redundant codification for single-bit self-correction.

is placed under that column of the array; if odd, a unit. For example, for the sixth bit, under column 6 we have

$$\begin{pmatrix}1\\1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}1\\0\\1\\1\end{pmatrix}=\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$$

$$\therefore r_6 = \bar{0}$$

for the seventh bit

$$\begin{pmatrix}1\\1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}1\\1\\1\\1\end{pmatrix}=\begin{pmatrix}1\\1\\0\\1\end{pmatrix}$$

$$\therefore r_7 = \bar{1}$$

and so forth.

Suppose that this redundant codification 1010 0101 is stored, and during the storage the third bit r_3 becomes changed. Then the incorrect code is 1011 0101. Of course we assume that we *know nothing* about the error in the codification except that there may be at most a single error. The problem remains to determine the correct bits I_0, I_1, I_2, and I_3 of the original information-carrying codification. The correct value of the bits I_1, I_2, I_3, and I_0 will each be derived, in this order, in spite of the error in the code. First note that, if $I_1 = 0$, then because of the method of formation we would clearly have $r_0 = r_1$, $r_2 = r_3$, $r_4 = r_5$, and $r_6 = r_7$; that is, adjacent pairs of bits would be the *same*. If $I_1 = 1$, then *none* of these adjacent pairs would be the same. A single error can affect only a single pair of positions; hence, to find I_1, we simply check to see whether we have more "same" pairs, or more "different" pairs. More same pairs means $I_1 = 0$; more different pairs means $I_1 = 1$. Similarly I_2 is determined from pairs (r_0,r_2), (r_1,r_3), (r_4,r_6), and (r_5,r_7) and I_3 from pairs (r_0,r_4), (r_1,r_5), (r_2,r_6), and (r_3,r_7) (see Fig. 11-13 for the work). To find I_0, consider an array composed of the rows of the original array corresponding to the I_1, I_2, or I_3 which are found to be

units and the redundant-code row. Under each column at this array place a zero if the number of units in the column is even, a unit if odd. If the number so formed is mostly zeros, $I_0 = 0$; if mostly units, $I_0 = 1$. Incidentally this new number clearly shows the position of the single error (see Fig. 11-14).

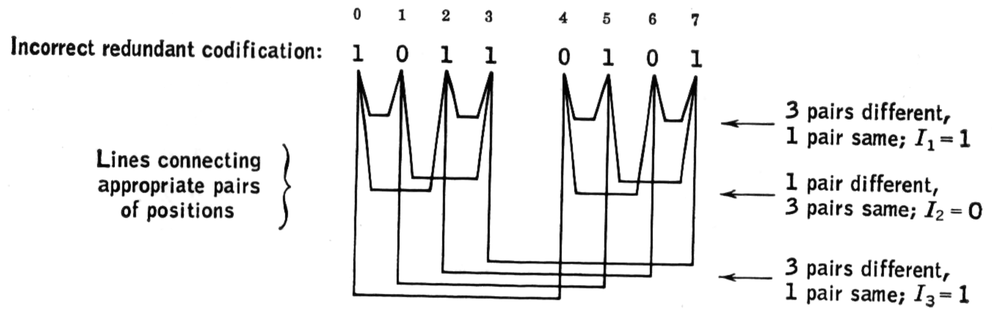


FIG. 11-13. Work for finding I_1, I_2, and I_3.

As another illustration, consider a 5-bit information-carrying codification and an extension of the above scheme to correct as many as 3 erroneous bits in the redundant codification. Let the original 5 bits be $I_0 = 0$, $I_1 = 0$, $I_2 = 1$, $I_3 = 1$, and $I_4 = 0$. Figure 11-15*a* represents the work required to generate the redundant codification which is to be transmitted or stored; the work proceeds just as in the example above. Suppose

	0123	4567	
$I_1 = 1$	0101	0101	
$I_3 = 1$	0000	1111	
Incorrect redundant codification	1011	0101	
New number	1110	1111	← more units; $\therefore I_0 = 1$

FIG. 11-14. Work for finding I_0. (Note that error occurred in position 3.)

that erroneous codification shown in Fig. 11-15 results from the storage; then the work required to determine the correct 5 information-carrying codification bits appears in Fig. 11-15*b*.

Observe that in the first example we used 8 bits to codify 4 information-carrying bits, or a redundancy of 4 bits; in the second example we used 16 bits to codify 5 bits, or a redundancy of 11 bits. In the first case we are trying to recognize 9 different possibilities, i.e., no error, or a single error in one of eight positions. This clearly takes 4 bits, the redundancy used. For the second case we are trying to recognize 697 different possibilities, i.e., no error, a single error in one of 16 positions, a double error that can occur in $\binom{16}{2} = 120$ different ways, or a triple

error that can occur in $\binom{16}{3} = 560$ different ways.† To distinguish among 697 possibilities, we clearly need 10 bits; so the 11-bit redundancy of the second example is one more than the minimum required. However, note that the 0th column of the array could have been omitted, and the scheme would work just as well. This would give us our minimum 10-bit redundancy.

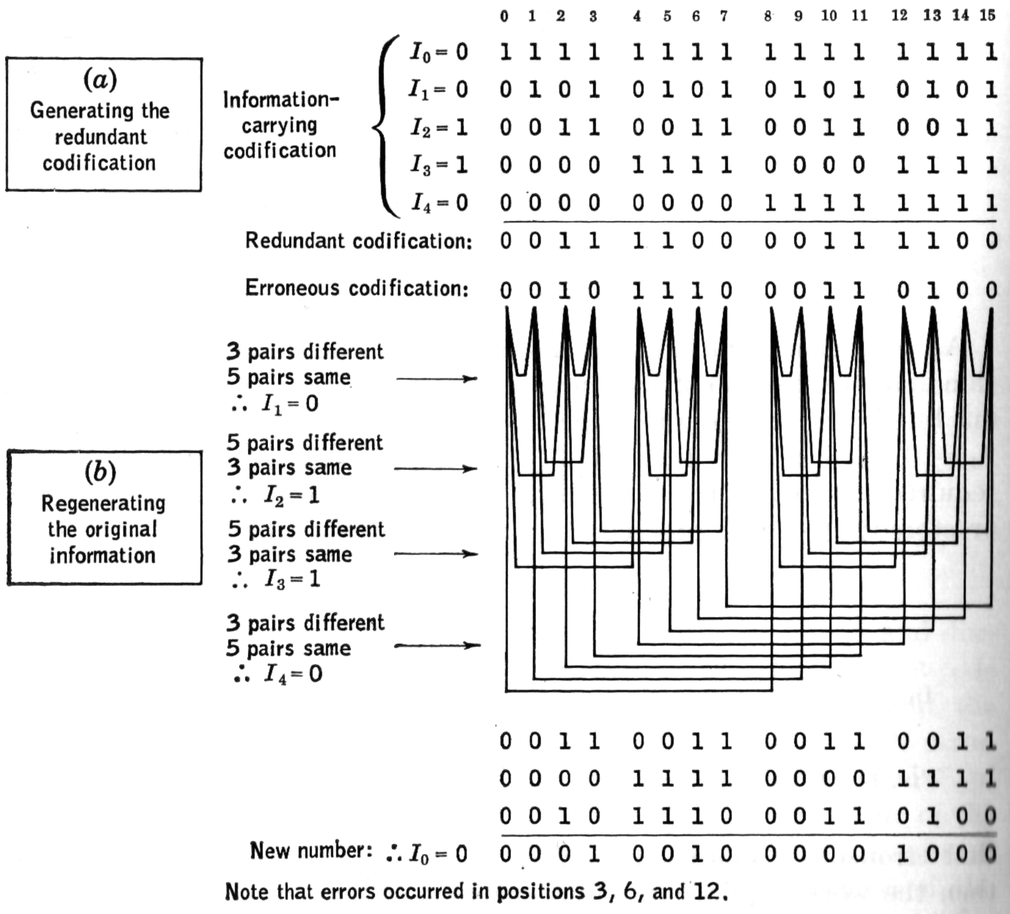


FIG. 11-15. Scheme illustrating three-error correcting process.

Superimposed Irredundant Codification. As we have seen, a binary codification that will distinguish uniquely between N characteristics requires at least $\log_2 N$ bit positions. Thus, if 4,500 characteristics are in the list, 13 bits are required. It frequently happens that, although N characteristics *could occur*, fewer than N of these *actually appear* in a particular situation. However, it is in general not known *beforehand*

† Recall that the number of ways of taking X objects Y at a time is given by the binomial coefficient $\binom{X}{Y} = \frac{X!}{Y!(X-Y)!}$.

how many will appear. In such cases, particularly when a *substantive codification for the characteristics is used*, at least $\log_2 N$ bit positions are used for the codification anyway, to take account of all eventualities. Thus, even though not all 4,500 characteristics will occur, 13 bits might still be reserved for the codification.

Furthermore, suppose that we are dealing with items each of which is associated with X of these characteristics; then, according to our above discussion, at least $X \log_2 N$ bit positions must be used to identify an item. For example, if $X = 3$ and $N = 4{,}500$, then $3 \times 13 = 39$ bit positions would be necessary. Thus, in principle, N^X items can be distinguished this way; but as we observed above, we are here considering situations where they do *not* all actually appear. Evidently we are being wasteful of bit positions because fewer than N^X items require fewer than $X \log_2 N$ bit positions for a codification. The problem therefore arises: *How can we use fewer than $X \log_2 N$ bits for our codification*, say $H < X \log_2 N$, *and yet maintain our ability to distinguish among all N characteristics?* The answer is found in the concept of superimposed codification.

By the *superimposed codification corresponding to X characteristics* we mean the result of forming the *logical sum* of the individual codifications for these X characteristics. That is, if the units represent, for example, notches on the edge of a card, two notches in the same position are equivalent to a single notch. Thus, if three characteristics have the following codes:

Characteristic	Position																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1st	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2d	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
3d	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0

then the corresponding superimposed codification is

0 0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 0 0 1 0

We shall illustrate by means of an example how superimposed codification can be made to satisfy the above criterion of using fewer than $X \log_2 N$ bits and yet maintain the ability to distinguish among all N characteristics. Suppose as above that $N = 4{,}500$. Let us codify the characteristics as follows: Use 20 positions, and let each codification contain precisely 4 units. This codification will allow us to code each characteristic uniquely, because 20 positions can be taken four at a time in

$\binom{20}{4} = \frac{20!}{4!(20-4)!} = 4{,}845$ different ways, and $4{,}845 > 4{,}500$. Finally, suppose that each item is associated with three characteristics; that is, $X = 3$. Then *let the codification of the item be the superimposed codifications of its associated three characteristics.* Note that the codification for an item now comprises only 20 positions, rather than the 39 thought necessary above, saving about half the positions.

The advantages of this technique can most readily be visualized in terms of the marginal punched cards described above, where each card represents an item, and where the marginal positions are the bit positions with a punch representing a unit, no punch a zero (see Fig. 11-16). Suppose that an item is associated with the three characteristics whose

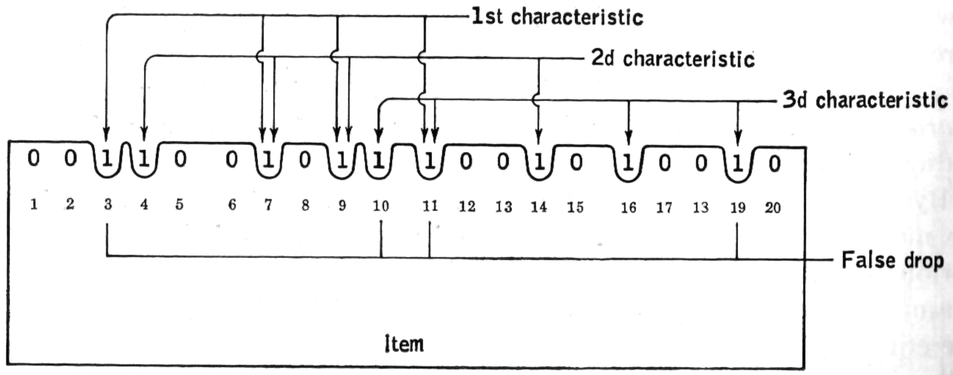


FIG. 11-16. Example of superimposed codification.

codifications appear above and that the 20 marginal positions of the item card are punched with the superimposed codification. Then, if all items associated with the first characteristic are being searched for, the sorting needles are placed in positions 3, 7, 9, and 11, and clearly the proper item cards will drop. Similarly for the second or third characteristic. However, *we must evidently pay a price for this convenience and saving in item-codification positions.* For suppose that the code for some other characteristic is

$$\overset{3}{00100} \qquad \overset{10}{00001} \qquad \overset{11}{10000} \qquad \overset{19}{00010}$$

Our item card (see Fig. 11-16) is *not* associated with this characteristic but nonetheless will drop in the search for items associated with this characteristic. The reason for this is that our codifications overlapped, allowing for more than three combinations of 4 units. In fact, in our illustrative example, nine punches appear, allowing for

$$\binom{9}{4} = \frac{9!}{4!(9-4)!} = 126$$

such possibilities, and hence $126 - 3 = 123$ possible *false* combinations. However, the situation is not so bad as it seems, for remember that we supposed that not all the $\binom{20}{4} = 4{,}845$ possible characteristics actually appear, and hence not all the false combinations will ever be sensed.

The subject of superimposed codification therefore resolves itself into the problem of determining the probability p that an item (card) not associated with a particular characteristic will be selected (dropped) during the search for that characteristic. We shall refer to p as the probability that an item (card) will be a *false drop*. The formulas given below for this probability p are based on the assumption that the codes for the characteristics are randomly chosen and that the associations of items with characteristics are also randomly chosen.

Let H = number of positions in superimposed codification
Y = number of units in a characteristic codification
X = number of characteristics associated with an item

Then
$$p = \frac{\left[\binom{H}{Y} - 1\right]^X - \sum_{i=0}^{i=Y-1} E_{Y-i}}{\binom{H}{Y}^X}$$

where

$$E_Y = \binom{Y}{0}\binom{H-Y}{Y}^X$$

$$E_{Y-1} = \binom{Y}{1}\binom{H-Y+1}{Y}^X - \binom{Y}{1}E_Y$$

$$E_{Y-2} = \binom{Y}{2}\binom{H-Y+2}{Y}^X - \binom{Y}{2}E_Y - \binom{Y-1}{1}E_{Y-1}$$

$$E_{Y-3} = \binom{Y}{3}\binom{H-Y+3}{Y}^X - \binom{Y}{3}E_Y - \binom{Y-1}{2}E_{Y-1} - \binom{Y-2}{1}E_{Y-2}$$

$$E_{Y-4} = \binom{Y}{4}\binom{H-Y+4}{Y}^X - \binom{Y}{4}E_Y - \binom{Y-1}{3}E_{Y-1} - \binom{Y-2}{2}E_{Y-2} - \binom{Y-3}{1}E_{Y-3}$$

and so forth, whence†

$$\sum_{i=0}^{i=Y-1} E_{Y-i} = \sum_{j=1}^{j=Y} (-1)^{j-1}\binom{H-j}{Y}\binom{Y}{j}$$

† This reduced form of the summation was developed by William R. Smith III.

For our example above, $H = 20$, $Y = 4$, $X = 3$; hence†

$$p = \frac{\left[\binom{20}{4} - 1\right]^3 - (E_1 + E_2 + E_3 + E_4)}{\binom{20}{4}^3}$$

$$= \frac{0.1131692 \times 10^{12} - 0.10890324 \times 10^{12}}{0.11323941 \times 10^{12}} = 0.0377$$

From probability theory we know that, if there are M items, then the expected number of false drops will be Mp. For our example, if $M = 200$,

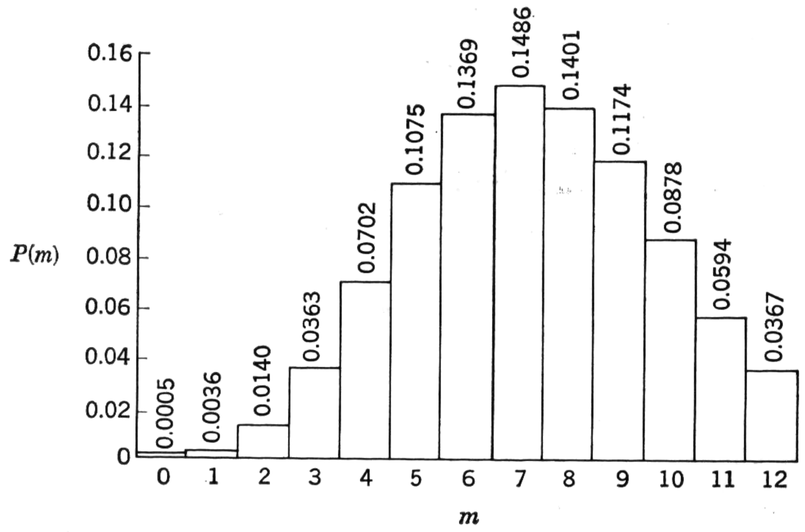


FIG. 11-17. Probability $P(m)$ of obtaining exactly m false drops for $H = 20$, $Y = 4$, $X = 3$, and $M = 200$ (here $h = 4$).

we would expect $200 \times 0.0377 = 7.54$ false-drop cards during the search on a particular characteristic. To find the probability $P(m)$ of obtaining m false drops, we use the formula from probability theory,

$$P(m) = \binom{M}{m} p^m (1 - p)^{M-m}$$

In Fig. 11-17 we have graphed some of these values for our example.

So far we have been considering false drops when searching on *one* characteristic only. Now we shall discuss the situation that arises when more than one characteristic is used in the search. Let n be the number of characteristics entering the search. The codifications of these particular characteristics may overlap. Let us suppose that their

† For numerical calculations some helpful results are $\binom{U}{0} = 1$, $\binom{U}{1} = U$, $\binom{U}{V} = \frac{U(U-1)\cdots(U-V+1)}{1 \times 2 \times \cdots \times V}$, $0! = 1$, $1! = 1$.

logical sum has precisely h units (punches). Then

$$p = \frac{\sum_{j=0}^{j=n-1} F_{n-j} - \sum_{i=0}^{i=h-1} E_{h-i}}{\binom{H}{Y}^X}$$

where

$$F_n = \binom{n}{0}\left[\binom{H}{Y} - n\right]^X$$

$$F_{n-1} = \binom{n}{1}\left[\binom{H}{Y} - n + 1\right]^X - \binom{n}{1}F_n$$

$$F_{n-2} = \binom{n}{2}\left[\binom{H}{Y} - n + 2\right]^X - \binom{n}{2}F_n - \binom{n-1}{1}F_{n-1}$$

$$F_{n-3} = \binom{n}{3}\left[\binom{H}{Y} - n + 3\right]^X - \binom{n}{3}F_n - \binom{n-1}{2}F_{n-1} - \binom{n-2}{1}F_{n-2}$$

and so forth, whence†

$$\sum_{j=0}^{j=n-1} F_{n-j} = \sum_{j=1}^{j=n} (-1)^{j-1}\left[\binom{H}{Y} - n - j\right]^X \binom{n}{j}$$

and

$$E_h = \binom{h}{0}\binom{H-h}{Y}^X$$

$$E_{h-1} = \binom{h}{1}\binom{H-h+1}{Y}^X - \binom{h}{1}E_h$$

$$E_{h-2} = \binom{h}{2}\binom{H-h+2}{Y}^X - \binom{h}{2}E_h - \binom{h-1}{1}E_{h-1}$$

$$E_{h-3} = \binom{h}{3}\binom{H-h+3}{Y}^X - \binom{h}{3}E_h - \binom{h-1}{2}E_{h-1} - \binom{h-2}{1}E_{h-2}$$

and so forth, whence†

$$\sum_{i=0}^{i=h-1} E_{h-i} = \sum_{j=1}^{j=n} (-1)^{j-1}\binom{H-j}{Y}^X \binom{h}{j}$$

Observe that, if $\binom{H}{Y} \gg n$, then $\sum_{j=0}^{j=n-1} F_{n-j} \approx \binom{H}{Y}^X$. This is the case for our example of $H = 20$, $Y = 4$, and $X = 3$. Let us suppose that we are searching on three characteristics with the codifications given above

† This reduced form of the summation was developed by William R. Smith III.

(see Fig. 11-16); then $h = 9$. Thus we have

$$p = \frac{\binom{20}{4}^3 - \sum_{i=0}^{i=8} E_{9-i}}{\binom{20}{4}^3} = \frac{(0.113732 - 0.113716) \times 10^{12}}{0.113732 \times 10^{12}} = 0.00014$$

It is interesting to compare $P(m)$ when searching on a single characteristic with $P(m)$ when searching on three characteristics that overlap to present the nine units of our illustration (see Exercise 11-26). Observe that h can range from 4 to 12; some values of p for various values of h are

h	4	6	9	12
p	0.0377	0.0061	0.00014	0.0000003

Note that, as long as $\binom{H}{Y} \gg n$, then the value of h is all-important and p is effectively independent of n. However, if, for example, $H = 6$, $Y = 2$, $X = 3$, then an n of 1, 2, or 3 can no longer be neglected.

11-8. Exercises

11-1. By means of the information given in Table 11-1 complete the lists and tables partially shown in Fig. 11-3*a* and *b*.

11-2. Using the information given in Fig. 11-1 and Table 11-1, compile Tabledex tables using as the characteristics the words themselves, in alphabetical order.

11-3. Using each of the three lists completed in Exercises 11-1 and 11-2, search for all items associated with each of the following sets of given characteristics: *evaluation, concept, design;* and *differentiation, application, England, gas.*

11-4. Assign population-based numbers to the items of Fig. 11-1, and compile the lists of the first searching method from the information of Table 11-1.

11-5. Draw a flow chart for each of the searching methods given in this chapter.

11-6. Punch a hole in each corner of sixteen 3 × 5 cards. Then write the names of the authors (see Fig. 11-1) on these cards, one name on each card. Mark off eight equally spaced positions on one 5-in. edge, nine positions on the other 5-in. edge of each card. Assign one word for each position, the same on each of the 16 cards. Next notch the edge of each card corresponding to the positions of words characteristic of this author's article (see Table 11-1). The cards can now be searched by placing a knitting needle under the deck of cards at the position of the desired word (see Fig. 11-18); the notched cards will drop, and the unnotched ones can be separated by putting another knitting needle through a corner hole of the raised cards. By appropriately separating collections of these cards as in Fig. 11-7, find all items associated with at least six, at least five, at least four, and at least three of the following seven given characteristics: *differentiation, application, England, evaluation, design, gas,* and *concept.*

11-7. Using the cards made in Exercise 11-6, find all items that are associated with *design* and *England* and as many of the following as possible: *differentiation, application, evaluation, gas,* and *concept.*

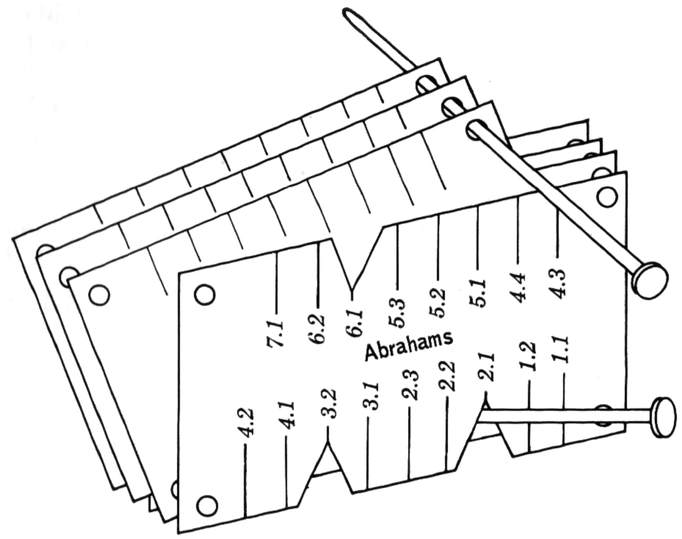


FIG. 11-18. Use of knitting needles to search marginal-punch cards.

11-8. Using the cards made in Exercise 11-6, find all items that are associated with either *evaluation* or *design* or both and as many of the following as possible: *differentiation, application, England, gas,* and *concept.*

11-9. How can the process of searching for all except one, except two, etc., be accomplished by means of the Tabledex tables? (HINT: A succession of tables must be used: one table for the first pass, two tables for the second pass, three tables for the third pass, etc.)

11-10. Draw a flow chart for a search for items associated with all the given characteristics except one, except two, etc. Assume the data to be given as in Fig. 11-2*b*. Then assume the data to be given in terms of the Tabledex tables, as in Fig. 11-2*c*.

11-11. Draw a flow chart for coding each of the four ordering methods of Sec. 11-5.

11-12. After shuffling a deck of playing cards, order them by each of the four methods of Sec. 11-5 without regard to suit (i.e., the four Queens, for example, will appear adjacent, after the Jacks and before the Kings, but not ordered by suit among themselves).

11-13. Suppose that one were ordering alphabetically by the fourth method, using a computer with 10 tape units. What radix could be used, and how would the tape units be utilized?

11-14. Draw a flow chart for coding a merge sort using four tapes, passing the numbers from two tapes to the other two.

11-15. Draw a flow chart for coding a merge sort, passing the numbers from m tapes to n tapes, where m and n can be specified arbitrarily.

11-16. What is the maximum number of passes required to put a deck of 52 cards in some order by suit as well as by denomination, merging from two bins to two bins? From three bins to one bin (counting the pass required to redistribute the cards into the three bins after each merge)? From four bins to four bins? From five bins to three bins? Considering these results, discuss the most efficient way to utilize a fixed number of tape units in a merge sort.

11-17. How many codifications can be made of no more than 10 positions with 26 symbols? If $K = 26$, what is the most efficient number of positions to use for codifying 100,000 words?

11-18. Given the information code 10101, find the redundant codification.

11-19. If a redundant codification was received as 0100 1000 0001 0011, what was the information code sent and in what positions did errors occur? Suppose that 0100 1000 0001 0001 was received; in what positions are the errors?

11-20. Suppose that there were 6 bits in an information code; if the scheme given above were used, how many errors could be detected and how many bits would there be in the redundant codification?

11-21. For $H = 4$, $Y = 2$, and $X = 2$, find p by means of the formulas of Sec. 11-7 for $n = 1$, $h = 2$; for $n = 2$, $h = 2$; for $n = 2$, $h = 3$; and for $n = 2$, $h = 4$. Check your result by making an enumeration of the $\binom{H}{Y}^X = \binom{4}{2}^2 = 36$ equally probable conditions.

11-22. Compute p for $H = 6, Y = 2, X = 3$ when searching on a single characteristic.

11-23. Compute p for $H = 6, Y = 2, X = 3$ when searching on three characteristics (that is, $n = 3$) that overlap so that $h = 5$.

11-24. Compare the results of Exercises 11-22 and 11-23 by graphing $P(m)$ for $M = 50$ in both cases.

11-25. Find the expected number of false drops for $H = 100$, $Y = 6$, $X = 4$ when $h = 15$.

11-26. Graph $P(m)$ for a search on three characteristics that overlap to present 9 units (i.e., $h = 9$, whence $p = 0.00014$—see above). Compare with Fig. 11-17.

11-27. *Irredundant Coding.* To our knowledge Mooers [see the References] first discussed irredundant coding. Formulas that differ from those given in this chapter are found in C. S. Wise, Mathematical Analysis of Coding Systems, in R. S. Carey and J. W. Perry (eds.), "Punched Cards," chap. 20, Reinhold Publishing Corporation, New York, 1951. The formulas given by Wise are not precise. (Why?) Prove the formulas given in this chapter of our text by means of an exact analysis of the problem.

REFERENCES

Sorting and Ordering

Bell, D. A.: The Principles of Sorting, *Computer J.*, vol. 1, no. 2, p. 71, July, 1958.

Davies, D. W.: Sorting of Data on an Electronic Computer, *Proc. IEE*, vol. 103B, suppl., p. 87.

Flores, I.: Analysis of Internal Computer Sorting, *J. Assoc. Computing Mach.*, vol. 8, p. 41, January, 1961.

Friend, E. H.: Sorting on Electronic Computer Systems, *J. Assoc. Computing Mach.*, vol. 3, p. 134, July, 1956.

Hosken, J. E.: Evaluation of Sorting Methods, *Proc. Eastern Joint Computer Conf.*, 1955, p. 35.

Isaac, E. J., and R. C. Singleton: Sorting by Address Calculation, *J. Assoc. Computing Mach.*, vol. 3, p. 169, July, 1956.

Shannon, C. E.: A Mathematical Theory of Communication, *Bell System Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.

Information Retrieval

Bracken, R. H., and H. A. Tillitt: Information Searching with a 701 Calculator, *J. Assoc. Computing Mach.*, vol. 4, p. 131, 1957.

Dennis, Bernard K.: Rapid Retrieval of Information, *Computers and Automation*, October, 1958, p. 8.

Digital Computer Newsletter, vol. 10, no. 4, p. 4, October, 1958.

Fairthorne, R. A.: Automatic Retrieval of Recorded Information, *Computer J.*, vol. 1, no. 1, p. 36, April, 1958.

Ledley, R. S.: TABLEDEX, A New Coordinate Indexing Method for Bound Book Form Bibliographies, *Proc. Intern. Conf. Sci. Inform.*, 1958, Area V, pp. 1221–1243, Washington, D.C.

Luhn, H. P.: A Statistical Approach to Mechanized Encoding and Searching of Literary Information, *IBM J. Research Develop.*, vol. 1, p. 309, 1957.

Mooers, C.: Putting Probability to Work in Coding Punched Cards, Zatocoding, *Zator Tech. Bull.* 3, 1947, Zator Company, Boston, Mass.; also Zatocoding and Development in Information Retrieval, *Aslib. Proc.*, vol. 8, no. 1, pp. 3–22, February, 1956.

National Science Foundation Booklet, "Current Research and Development in Information Retrieval."

National Science Foundation Booklet, "Non-conventional Technical Information Systems in Current Use."

Office of Research and Development, U.S. Patent Office: "Patent Office Research and Development Reports," U.S. Department of Commerce.

Proc. Intern. Conf. Sci. Inform., 1958, Area V, pp. 817–1268.

Ray, L. C., and R. A. Kirsch: Finding Chemical Records by Digital Computers, *Science*, vol. 126, p. 814, 1957.

Shera, J. H., A. Hert, and J. W. Perry: "Information Systems in Documentation," Interscience Publishers, Inc., New York.

Waldo, W. H., and M. DeBacker: Printing Chemical Structures Electronically: Encoded Compounds Searched Generically with IBM 702, *Proc. Intern. Conf. Sci. Inform.*, 1958, Area IV, pp. 711–730.

Wildhack, W. A., and J. Stern: "The Peek-a-boo System in the Field of Instrumentation," p. 209, Interscience Publishers, Inc., New York.

Error-detecting and -correcting Codes. The method given in this chapter for constructing error-correcting codes is essentially that of Irving S. Reed (see A Class of Multiple-error-correcting Codes and the Decoding Scheme, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 34–49, September, 1954). The initial theory was first stated in a well-known paper of R. W. Hamming (see Error Detecting and Error Correcting Codes, *Bell System Tech. J.*, vol. 29, pp. 147–160, April, 1950). Many other papers have recently appeared in this field, and we list some of these below. It would be well to consider these papers after Chap. 10 of the text has been mastered.

Brown, A. B., and S. T. Myers: Evaluation of Some Error Correction Methods Applicable to Digital Data Transmission, *IRE Conv. Record*, vol. 6, pt. 4, pp. 37–55, 1958.

Elias, P.: Error Free Coding, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 29–38, September, 1954.

Golay, M.: Binary Coding, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 23–28, September, 1954.

Huffman, D.: A Linear Circuit Viewpoint on Error Correcting Codes, *IRE Trans. on Inform. Theory*, vol. IT-2, pp. 203–234, September, 1956.

Peterson, W.: An Experimental Study of a Binary Code, *Communs. and Electronics*, July, 1958, pp. 388–392.

Sacks, G.: Multiple Error Correction by Means of Parity Checks, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 145–147, December, 1958.

Shannon, C. E.: General Treatment of the Problem of Coding, *IRE Trans. on Inform. Theory*, vol. IT-1, pp. 102–104, February, 1953.

Shapiro, H. S., and D. L. Slotnik: On the Mathematical Theory of Error-correcting Codes, *IBM J. Research Develop.*, January, 1958, pp. 25–34.

Slepian, D.: A Class of Binary Signaling Alphabets, *Bell System Tech. J.*, January, 1956, pp. 203–234.

Ulrich, W.: Non-binary Error Correction Codes, *Bell System Tech. J.*, November, 1957, pp. 1341–1388.

An informative book on this subject which also includes an extensive bibliography is:

Peterson, W. W.: "Error-correcting Codes," The M.I.T. Press and John Wiley & Sons, Inc., New York, 1961.

APPENDIX

The purpose of this appendix is to present a sampling of instruction types from three-, two-, and one-address commercially available computers. The instructions described do not at all represent the repertoire of these computers, but rather just some sample instructions. The sample instructions were *not* chosen to reflect the versatility or scope of the respective computers, but rather were chosen primarily to illustrate some of the principles expounded in the text (see Secs. 3-1, 3-2, 3-4, 3-5, and 3-8). Also, in some cases we have simplified the description of the instructions for pedagogical purposes. For detailed treatments of these instruction systems, see the respective programming manuals referred to below.

Honeywell 800:† *Example of a Three-address Instruction System.* The Honeywell 800 computer utilizes a three-address instruction system (see Fig. A-1) that also includes relative, indirect, compound, and augmented addressing techniques (see Sec. 3-5). Figure A-2 summarizes the alternative possibilities, which for each address in the instruction are distinguished by three bits, called X, Y, and Z. The X bit is in position 4, 5, or 6 of the operation code, corresponding to the α, β, or γ address, respectively; the Y bit is in position 1 of the α, β, or γ address itself; and the Z bit is in position 7 of the α, β, or γ address itself, but is not utilized for direct and simple-relative addressing. There can be up to 16 banks of core memory, each bank of 2,048 words; there are eight banks of registers (which constitute the so-called "special memory"). Augmented addressing is utilized for the selection of the appropriate memory and register banks, as will be described. As seen in Fig. A-2, six types of addressing are available, four for addressing the memory and two for addressing the registers, as follows:

(*a*) DIRECT MEMORY ADDRESSING. The 11-bit address is augmented on the left by the four most significant bits of the current-address register, to produce a complete 15-bit address in the same memory bank as the instruction.

† See "Honeywell 800 Programmers' Reference Manual," Minneapolis-Honeywell DATAmatic Division, Wellesley Hills, Mass., 1960.

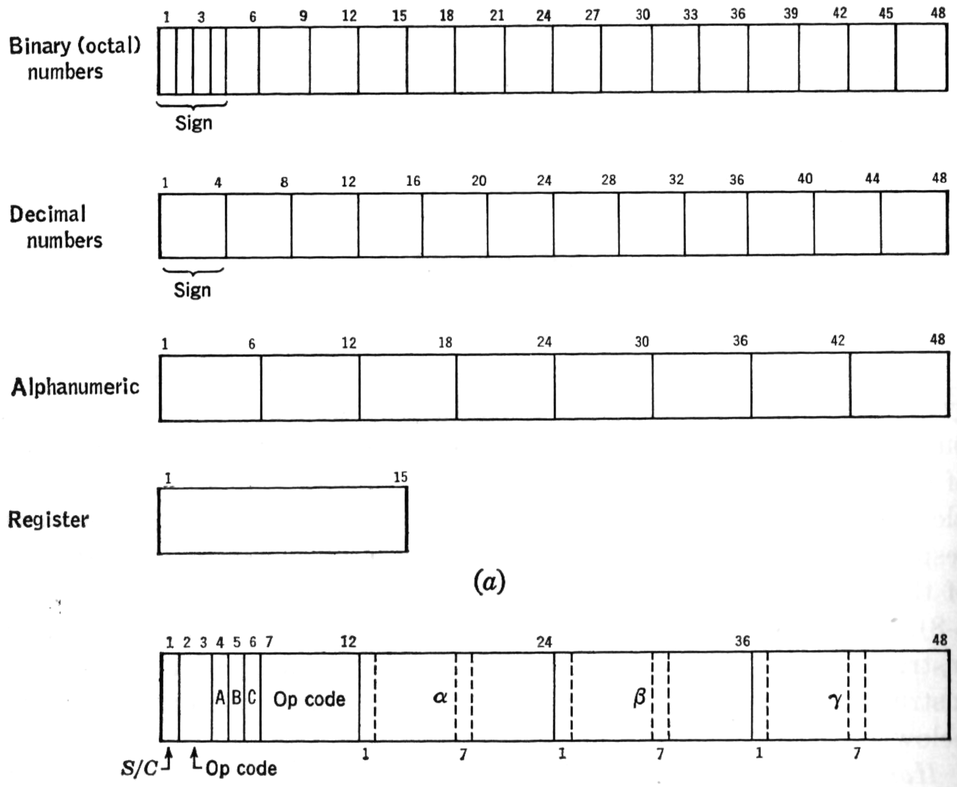


FIG. A-1. Number and instruction formats for the Honeywell 800 computer. In the instruction format the bit in position 1 denotes which of two possible current-address registers (S or C) is to be used for the address of the next instruction.

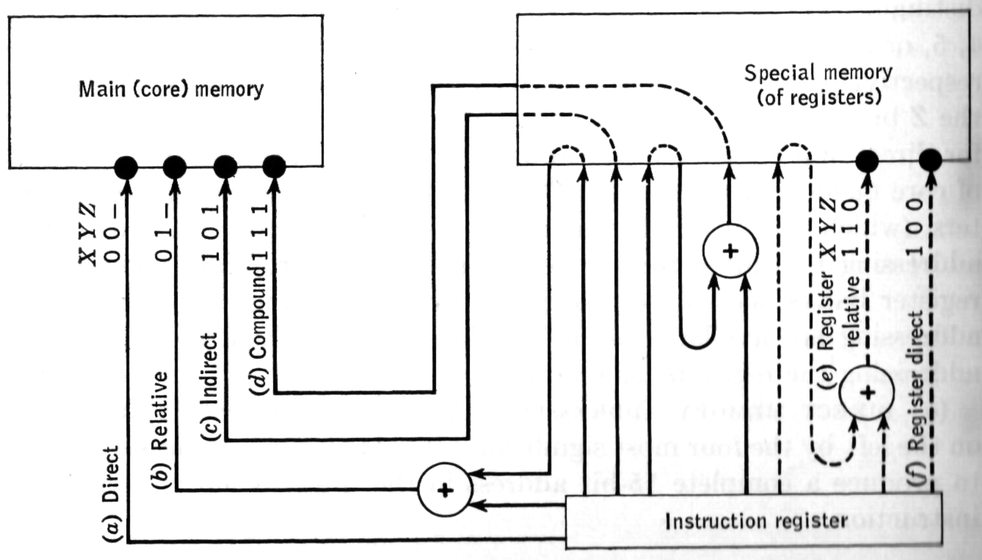


FIG. A-2. The six methods of addressing the Honeywell 800 computer (see text for descriptions).

(*b*) RELATIVE MEMORY ADDRESSING. Bits 2, 3, and 4 of the address select one of eight registers; these bits are, however, first augmented on the left by the three most significant bits of the current-address register so as to select the proper bank of registers. Then the contents of this 15-bit register is added to the remaining 8 bits of the address to form the desired relative address. Note that this final address can be in a *different memory bank* from that of the instruction (why?).

(*c*) INDIRECT MEMORY ADDRESSING. Bits 8 through 12 of the address are augmented by the three most significant bits of the current-address register, to select one of 32 registers in the register bank which corresponds to the location of the instruction being executed; the contents of this register becomes the desired address. As in relative memory addressing, the final address can be in any memory bank. As an added feature, after the core-memory access has been completed, bits 2 through 6 of the address are added to the contents of the selected register as an increment, and the sum is replaced in the register (resulting in register incrementation).

(*d*) COMPOUND MEMORY ADDRESSING. Bits 2 through 4 of the address in the instruction being executed are augmented by the three most significant bits of the current-address register, to select one of eight registers in the register bank of the instruction; we shall call the register so selected the *relative register*. Bits 5 through 15 of this relative register are added to bits 5 through 12 of the address in the instruction, "right-justified" (i.e., bit 15 is aligned with bit 12, 14 with 11, etc.); the resultant sum is augmented with bits 2 through 4 of the relative register, completing a 14-bit "relative" address in positions 2 through 15. Eight of these 14 bits, namely, bits in positions 2 through 4 and 11 through 15, select another register, which we shall call the *indirect register*. The contents of the indirect register forms the address of the desired word in the core memory, completing the compound indexing process. Of the six remaining bits of the "relative" address, the bits in positions 5 through 9 are added (right-justified) to the contents of the indirect register as an increment (register incrementation) after the core-memory access has been completed. The bit in position 10 should be the original Z bit, and hence care must be taken that it is not changed from 1 in the formation of the "relative" address.

(*e*) RELATIVE REGISTER ADDRESSING. This is similar to compound memory addressing in that a relative register is first chosen and then another register is selected from bits in positions 2 through 4 and 11 through 15 of the "relative" address. However, here the selection of this latter register is the desired result. Care must be taken that the Z bit, located in position 10 of the "relative" address, this time remains 0.

(*f*) DIRECT REGISTER ADDRESSING. Bits 11 through 15 of the address in the instruction are augmented by the three most significant bits of the current-address register to form the address of the desired register. As an added feature, bits 2 through 7 are added to the contents of the register selected *after* such contents has been utilized by the instruction (resulting in register incrementation).

The following are typical instructions of the Honeywell 800 (with the manufacturer's symbolic operation codes shown in square brackets):

Arithmetic instructions
- $(\alpha) + (\beta) \rightarrow \gamma$ [BA]
- $(\alpha) - (\beta) \rightarrow \gamma$ [BS]
- $(\alpha) \times (\beta) \rightarrow \gamma$ [BM]
- $(\alpha)/(\beta) \rightarrow \gamma$ [BD]

Transfer

Transfer (α) into β; take (γ) as next instruction [TS].

Comparison

If $(\alpha) \neq (\beta)$, take (γ) as next instruction; otherwise continue as usual [NN].

Accumulate

Here α must be indirect or compound, utilizing register incrementation. The sum is formed of β numbers, starting with the contents of the word specified by α and continuing with the addresses formed through incrementing; then the result is placed in γ [DT].

Multiword transfer

Transfer β words from successive positions starting at α to successive positions starting at γ [TN].

Logical instructions (where $\cdot$, $+$, and $\bar{}$, but *not* $\rightarrow$, are logical operations)
- $(\alpha) \cdot \overline{(\beta)} + \overline{(\alpha)} \cdot (\beta) \rightarrow \gamma$ [HA]
- $(\alpha) + (\beta) \rightarrow \gamma$ [SM]
- $(\alpha) \cdot (\beta) \rightarrow \gamma$ [EX]

Masked instructions

In order to preserve part of the result word [usually (γ)] after an operation is performed, a so-called "mask" word can be used. Most often, the bits in the result word corresponding to the zero bits in the mask word remain unchanged. The address of the mask word is composed by bits from the instruction augmented by bits from a special *mask register;* the mask register need not be addressed since it is automatically referenced during the execution of mask-utilizing operations. The mask register can be loaded in the same manner as any other register (see parts *e* and *f* above). A typical masked instruction is a right-circular shift: shift the contents of α right (around to the beginning of the word) by the number of positions specified by bits 25 through 30 (in β) and put the result into γ, but preserving those bits of γ corresponding to the zeros of the mask word; the mask word is found as the contents of the address composed by augmenting bits 31 through 36 of the shift instruction with bits 2 through 10 of the mask register of the same register bank as the instruction [SPS].

Univac 1103A Scientific:† *Example of a Two-address Instruction System.* The Univac Scientific computer has a two-address format (see Fig. A-3); it has only a single addressable register, the *Q* register, which

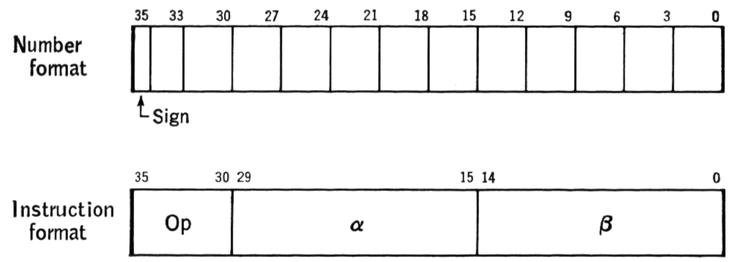


FIG. A-3. Number and instruction formats for the Univac 1103A computer.

cannot be used, however, for relative indexing. Typical instructions (with the manufacturer's symbolic operation codes shown in square brackets) are:

Arithmetic instructions
- $(\alpha) + (\beta) \rightarrow \alpha$ [RA]
- $(\alpha) - (\beta) \rightarrow \alpha$ [RS]
- $(\alpha) \times (\beta) \rightarrow$ acc [MD]
- $(\alpha)/(\beta) \rightarrow \beta$ [DV]
- $(\alpha) + (Q)(\beta) \rightarrow$ acc [MA]
- $(\alpha) \times (Q) + (\beta \rightarrow$ Q [FP]

Logical instructions (where $\cdot$ and $\bar{}$, but *not* $+$ and $\rightarrow$, are logical operations)
- $(\alpha) \cdot (Q) \rightarrow \beta$ [QT]
- $(\alpha) \cdot (Q) + (\beta) \cdot \overline{(Q)} \rightarrow \beta$ [QS]
- $(\text{acc}) + (\alpha) \cdot (Q) \rightarrow \beta$ [QA]

Here Q acts as a mask for the operand which it logically multiplies.

Repeat instruction

Repeat the following instruction the number of times given by positions 15 through 26. If bit 27 is a unit, then the remaining portion of α is increased by 1 after each execution of the following instruction; if bit 28 is a unit, then the same is done for β [RP].

† See "Univac 1103A Programming Manual," Remington Rand Univac Division of Sperry Rand Corporation, Philadelphia, 1958.

Comparisons and jumps

If (α) > (acc) take β [TJ]; replace (α) with $C + 1$ in its β positions, where C is the contents of the current-address register; then take β [RJ].

Other instructions

Normalize (α), and store the number of positions shifted in (β), with the normalized result found as (acc) [SF]; transmit (α) to β [TP]; transmit the α address of (α) to the α address of (β) [TU]; similarly transmit for β addresses [TV].

IBM 7090:† Example of a One-address Instruction System. The IBM 7090 has a one-address instruction system (see Fig. A-4) and includes three index registers for *relative addressing*. Bits in positions 18, 19, and 20 address these three index registers, the addresses of which are respectively 001, 010, and 100. The relative address is found by *subtracting* the contents of the designated index register from α. The index registers can themselves be loaded and modified by special instructions, as will be mentioned below. Reference to two or more index registers at the same time, i.e., by loading positions 18, 19, and 20 with 110, 101, 011, or 111, results in the use of a number obtained by forming the logical sum of the register contents indicated for the relative addressing. *Indirect addressing* is also possible, and is indicated by placing 11 in positions 12 and 13 of the instruction: here the direct address specified by the instruction (whether or not relative) is interpreted as the location in the core memory of the word whose contents (i.e., bits 21 through 35) is the final desired indirect address; this final desired indirect address may *itself* be a relative address if so specified in positions 18, 19, and 20 of the location where it was found. The use of relative addressing in conjunction with indirect

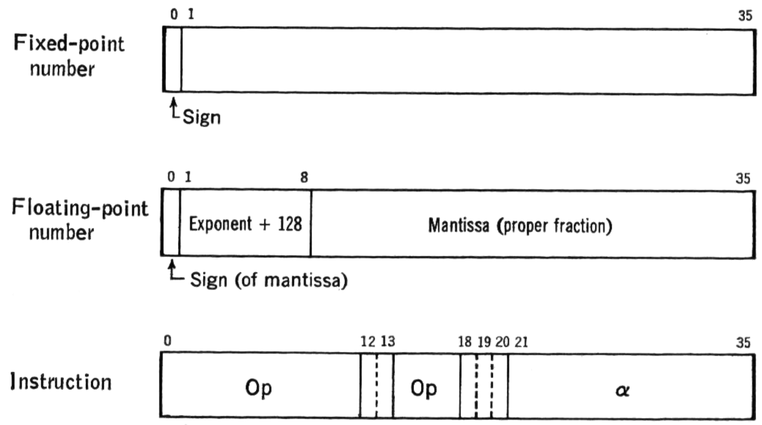


FIG. A-4. Number and instruction formats for the IBM 7090 computer.

† See "Reference Manual IBM 7090 Data Processing System," International Business Machines Corporation, Data Processing Division, White Plains, N.Y., 1960.

addressing, as shown in Fig. A-5, has been termed compound addressing (see Sec. 3-5).

Typical instructions (with the manufacturer's symbolic operation codes shown in square brackets) are the following:

Arithmetic instructions

(α) + (acc) → acc [ADD]

(acc) − (α) → acc [SUB]

(α) → acc [CLA]

For multiplication, the MQ register, which is simply a right extension of the accumulator, is utilized, and the double-length product is stored in both MQ and the accumulator. The MQ register is loaded by (α) → MQ [LDQ], and its contents is stored by (MQ) → α [STQ]. The

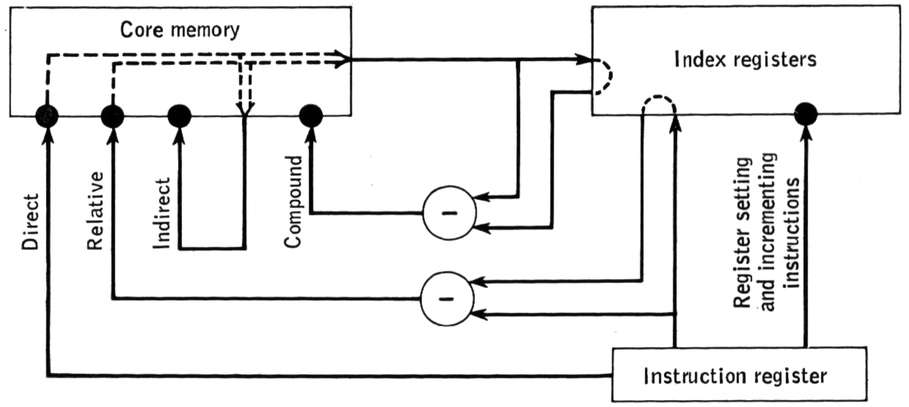


FIG. A-5. Addressing methods in the IBM 7090 computer (see text for descriptions).

multiply instruction becomes (MQ) × (α) → acc-MQ [MPY]. For division a double-length divisor stored in acc-MQ is divided by (α), with the resulting quotient appearing in MQ, and the remainder in acc. The *divide* instruction becomes (acc-MQ)/(acc) → MQ [DVH]. There are also floating-point arithmetic operations, both normalized and unnormalized.

Word-manipulation instructions

Shift (acc) left α positions [ALS]; shift (acc-MQ) left α positions [LLS]; (acc) → α [STO]; the location of this instruction +1 is stored in bits 21 to 35 of (α) [STL]; (acc) → MQ and (MQ) → acc [XCL]; take α [TRA]; if (acc) is positive, take α [TPL]; if (acc) is negative, take α [TMI]; if (acc) > (MQ), take α [TLQ].

Index-manipulation instructions

Place the logical complement of the current address in the index register specified in positions 18 through 20, and take α [TSX]; add the number in positions 3 through 17 to the specified index register, and take

α [TXI]; if the contents of the specified index register is greater than the number in positions 3 through 17, take α [TXH] (NOTE: In these last two instructions the operation code is located in positions 0 through 2 of the instruction); transfer the address part of (α) to the specified index register [LXA]; put α into the specified index register [AXT]; transfer the contents of the specified index register into the address part of (α) [SXA].

Logical instructions (where $\cdot$, $+$, and $^-$ represent logical operations)

$(\text{acc}) + (\alpha) \rightarrow \text{acc}$ [ORA]
$(\text{acc}) + (\alpha) \rightarrow \alpha$ [ORS]
$(\text{acc}) \cdot (\alpha) \rightarrow \text{acc}$ [ANA]
$(\text{acc}) \cdot \overline{(\alpha)} + \overline{(\text{acc})} \cdot (\alpha) \rightarrow \text{acc}$ [ERA]
$\overline{(\text{acc})} \rightarrow \text{acc}$ [COM]

Sense-indicator instructions

These operations, which are associated with the use of a special sense-indicator register, denoted by SI, serve to load, store, and logically change the contents of the register (where $\cdot$, $+$, and $^-$ represent logical operations): $(\alpha) \rightarrow \text{SI}$ [LDI]; $(\text{SI}) \rightarrow \alpha$ [STI]; $(\text{SI}) + (\alpha) \rightarrow \text{SI}$ [OSI]; $(\text{SI}) \cdot \overline{(\alpha)} \rightarrow \text{SI}$ [RIS]; $(\text{SI}) \cdot \overline{(\alpha)} + \overline{(\text{SI})} \cdot (\alpha) \rightarrow \text{SI}$ [IIS]; if $\overline{(\text{SI})} \cdot (\text{acc})$ contains all zeros, take α [TIO]; if $(\text{SI}) \cdot (\text{acc})$ contains all zeros, take α [TIF]. In all except the straight transfer operations, (α) can be considered as a *mask*, with only those positions of (SI) considered which correspond to units in (α) (why?).

It is instructive to note that the 7090 one-address system requires special instructions to handle the index registers, whereas the same effect is obtained in the 800 three-address system by utilizing the capability for directly handling the index registers in the same manner as core words. Also, the indirect addressing in the 800 system is accomplished through the very high speed special memory of index registers, whereas the 7090 system requires intermediate access to the slower core memory.

NAME INDEX

Abhyankar, S., 494
Abramowitz, M., 429*n.*
Ackerman, W., 493
Ackoff, R. L., 292
Adams, C. W., 157, 291
Aikin, H., 6
Alexander, S. N., 6
Allen, L. E., 494
Allutisi, E. A., 381
Alt, F. L., 49, 433
Amarel, S., 381
Andrews, A. J., 51
Angle, C., 291
Antosiewicz, H. A., 50
Arden, B. W., 238, 378
Assuer, A., 51
Astrahan, M. M., 124

Babbage, C., 2
Backus, J., 238
Baggott, B. A., 434
Bar-Hillel, Y., 495*n.*
Baran, P., 381
Barlow, J. S., 50
Barnett, M. P., 377
Bartlett, F. C., 381
Barton, R. S., 192
Baxendale, P. B., 381
Bean, K., 52
Bell, D. A., 534
Bellman, R., 49, 333*n.*, 378
Bemer, R. W., 192
Berman, M., 50
Bermick, M. D., 238
Bernstein, A., 379, 493
Bigilo, J., 6
Blaauw, G. A., 125
Boole, G., 453, 493
Booth, A. D., 49
Booth, K. H. V., 49
Bottenbruch, H., 238, 379
Bowden, B. V., 49
Boyer, P. W., 292
Bracken, R. H., 534
Brooks, F. P., Jr., 125
Brown, A. A., 292
Brown, A. B., 535
Buchholz, W., 125
Bush, G. P., 292
Bush, V., 156

Cahn, L., 51
Callender, E. D., 238
Cameron, S., 382
Canning, R. G., 49, 292
Cannon, E. W., 51
Cantrell, H. N., 238
Carey, R. S., 534
Carnap, R., 494
Carr, J. W., 377
Carroi, P., 292
Chang, J. J., 382
Chapin, N., 49
Chomsky, N., 194*n.*, 238, 380
Chu, J. C., 125
Church, A., 379, 493
Churchman, C. W., 292
Clark, E., 378
Clark, W. A., 50, 381
Claydon, J. B., 51
Clymer, A. B., 51, 52
Cocke, J., 125
Cohen, M., 493
Cole, J. S., 292
Cole, K. S., 50
Coles, G. R., 380
Copilowish, I. M., 493
Coveland, D. W., 380
Culbertson, J. T., 381
Curtiss, J. 6

Davidson, L., 494
Davies, D. W., 534
DeBacker, M., 535
De Morgan, A., 440
Dempsey, J. M., 52
Denman, H. H., 402*n.*
Dennis, B. K., 534
D'Esopo, D. A., 52
Dickie, H. F., 52
Dinneen, G. P., 381
Dixon, H. L., 52
Donan, J. F., 156
Douglas, A. S., 125
Doyle, W., 381
Dreyfus, P., 157
Drinnan, J. H., 51
Dunwell, S. W., 157

Eckert, J. P., Jr., 6, 49, 125, 157
Edmundson, H. P., 49, 381
Eisenhardt, R. D., 157
Elbourn, R. D., 460*n.*
Elias, P., 535
Ellis, T. O., 125, 380
England, D., 378
Estrin, G., 381
Evans, A., Jr., 378

Fairthorne, R. A., 535
Farley, B. G., 50, 381
Feigenbaum, E. A., 378, 380
Feurzeig, W., 238
Firstman, S. I., 292
Fleishman, A. M., 52
Flores, I., 534
Floyd, R. W., 189*n.*, 378
Forrester, J., 6
Forsythe, G. E., 49
Fredkin, E., 125
Friedberg, R. M., 380
Friend, E. H., 534
Frigenbaum, E. A., 239
Fritz, F. H., 292

Gallagher, J. D., 292
Galler, B. A., 238, 378
Gass, Saul I., 49, 379
Geisler, M. N., 292
Gelernter, H. L., 192, 349*n.*, 377, 380
Gerberich, C. L., 192, 377
Germond, H. H., 49
Gilli, S., 50
Gilmore, J. W., 494
Gilmore, P., 380
Golay, M., 535
Goldstine, H., 6
Goodman, R., 192
Gorn, S., 378
Gotlieb, C. C., 49
Grabbe, E. M., 433
Graham, R. M., 238, 378
Grau, A. A., 238
Green, B. F., Jr., 239, 378, 380
Green, J., 238, 378
Greenwald, I. D., 192
Gregory, R. H., 49, 292

Grimsdale, R. L., 381
Guillemin, E. A., 384*n.*

Haddad, J. A., 6
Hamming, R. W., 535
Hansen, J. R., 192, 378, 380
Harman, H. H., 52
Harris, B., 460*n.*, 494
Harris, Z. S., 238
Hartree, D. R., 156
Haskins & Sells, 292
Hastings, C., Jr., 402*n.*, 434
Hattery, L. H., 292
Hebb, D. O., 381
Henrici, P., 389*n.*
Hert, A., 535
Hetherington, A. W., 51
Hetrick, J. C., 292
Highleyman, W. H., 381
Hilbert, D., 435, 493
Hildebrand, F. B., 434
Hildebrandt, P., 292
Hiller, L. A., Jr., 371*n.*, 375*n.*, 381
Hindle, R., 292
Holland, J. H., 125
Hopper, G. N., 192, 292
Hosken, J. E., 534
Householder, A. S., 49, 434
Hovland, C. I., 50, 379, 381
Howard, R. A., 49, 379
Huffman, D., 535
Hume, J. N. P., 49
Hunt, E. B., 50, 381
Huntington, E. V., 494
Huskey, H. D., 238
Huxley, J., 1, 2

Ingerman, P. Z., 238, 378
Irons, E. T., 238, 299*n.*, 378
Isaac, E. J., 125, 534
Isaacson, L. M., 371*n.*, 375*n.*, 381
Isbitz, H., 292
Issacson, E., 50

Jacquard, J. M., 2
Jenkins, H. M., 379
Jevons, W. S., 494
Jones, Rebecca, 49
Jones, W. T., Jr., 292
Jourdain, P. E. B., 493

Kalsky, H. G., 125
Kamentsky, L. A., 381
Karnaugh, G., 494
Karp, H. R., 51, 292
Kavanagh, T. F., 192
Keenoy, C. L., 292
Keller, H., 381
Kemeny, J. G., 379
Kibbee, J. M., 293
Kilburn, T., 125, 381
King, F. E., 238
King, J., 238
Kircher, P., 293
Kirsch, R. A., 51, 535
Kister, J., 380
Kleene, S. C., 187*n.*, 379
Knuth, D. E., 239
Kochen, M., 380
Kozmetsky, G., 293
Kramer, S. J., 293
Kruse, B., 293

Laden, H. A., 293
Lambek, J., 238
Lanczos, C., 434
Laplace, P. S., 417, 418
Lassing, L. P., 293
Laudbach, P. B., 293
Lawrence, D. H., 380
Lawson, R. H., 378
Ledley, R. S., 49–51, 125, 334*n.*, 378, 379, 476*n.*, 478*n.*, 480*n.*, 494, 535
Lee, T-H., 49, 293
Lefkowitz, B., 52
Leibnitz, G. W. v., 2
Leveson, J. H., 293
Levin, H. S., 49, 293
Levy, C., 434
Lewis, D. C., 293
Lewis, T. S., 52
Livesley, R. K., 49, 51
Locke, W. N., 49
Lubkin, S., 6
Luce, R. D., 325*n.*, 475*n.*
Luhn, H. P., 381, 535
Lusted, L. B., 51, 334*n.*, 379

McCarthy, J., 192, 378, 380
McCluskey, E. J., Jr., 454*n.*, 494
McCormick, E. M., 49
McCracken, D. D., 49, 293
McCulloch, W. S., 382
McDermio, W. L., 125
McIlroy, M. D., 192
Magnus, K., 379
Malcolm, D. G., 52
Mandl, M., 50
Marden, E., 51
Marlow, W. H., 140*n.*
Marton, F. D., 51, 157
Mason, H. L., 51
Mauchly, J., 6
Mealy, G. H., 239, 378
Mendelson, M. J., 157
Merner, J. N., 238
Miller, G. A., 192, 381
Milne, W. E., 400*n.*, 408*n.*, 434
Milne-Thomson, L. M., 394*n.*
Minor, R., 293
Minsky, M., 362*n.*, 380, 381
Mirkel, H., 379
Mooers, C., 534, 535
Moore, C. J., 52
Muns, F. H., 293
Myers, S. T., 535

Nagel, E., 493
Napier, J., 2
Naur, P., 238
Neisser, U., 382
Newell, A., 125, 239, 351*n.*, 355, 378, 380, 381
Newman, J. R., 2*n.*
Newton, I., 2
Nielsen, K. L., 434

Oettinger, A. G., 380
Oliver, F. F., 293
Ornstein, G. N., 51
Oswald, V. A., 378, 382

Palevsky, M., 157
Pardee, D. R., 293
Parsons, J. R., 51
Pascal, B., 2
Paterson, H. E., 125
Pattishall, E., 51
Peck, L. G., 292
Perlis, A. J., 192, 239, 378
Perry, J. W., 534, 535
Perry, M. N., 52
Peterson, W., 535, 536
Pipberger, H. V., 51
Pitts, W., 382
Plugge, W. R., 52
Polya, G., 379
Potts, T. F., 51
Price, Waterhouse & Co., 293

Quine, W. v. O., 454*n.*, 493, 494

Rabinowitz, P., 50
Raiffa, H., 325*n.*
Rajchman, J. A., 125
Ramo, S., 433
Rath, G. J., 382
Ray, C., 51, 535
Reed, S. I., 535
Reichenbach, H., 436, 493, 494
Reid, M., 51
Resnick, A., 382
Roberts, L. G., 382
Rochester, N., 6, 349*n.*, 380
Rosenblatt, F., 382
Ross, P. T., 378
Rosser, J. B., 493
Roth, J. P., 494
Rotkin, L., 293
Rowe, A. J., 293
Runge, C., 413, 414
Ruskin, V. W., 51
Russell, B., 435, 493

Saber, N., 239, 378
Sacks, G., 535
Salveson, M. E., 293
Samelson, K., 239
Samuel, A. L., 380
Sanford, J. R., 238
Sattley, K., 239
Savage, T. R., 382
Scarborough, J. B., 434
Schmitt, W. F., 125
Schoenfeld, R. L., 50
Schroeder, E., 435, 444, 494
Scott, A. E., 293
Selfridge, O. G., 382
Semment, J. E., 293
Shannon, C. E., 534, 536
Shapiro, H. S., 536
Shaw, J. C., 125, 239, 351*n.*, 355, 380, 381
Sheffer, H. M., 442
Shepard, R. N., 379, 382
Shera, J. H., 535
Sherman, H., 382
Shubik, M., 52
Silber, W. B., 157
Simon, H. A., 125, 295*n.*, 351*n.*, 355, 380, 381
Simpson, H. R., 51, 413, 414
Singleton, R. C., 125, 534
Slepian, D., 536
Slotnik, D. L., 536
Slutz, R., 6
Smith, H. J., 192
Smith, J. W., 239, 378
Smith, W. R., III, 476*n.*, 529*n.*, 531*n.*
Snell, J. L., 379
Soroka, W. W., 52
Sperry, R. W., 382
Sprague, R. E., 157
Steel, T. B., 293
Steger, W. A., 292
Stegun, I. A., 429*n.*
Steinbunch, K., 378
Stephenson, R., 379
Stern, J., 535
Stibitz, G., 6
Stoker, J. J., 50
Stoller, D. S., 293
Stratton, J. A., 81*n.*
Sumner, G. H., 381
Swanson, J. A., 125
Swanson, P. R., 52, 293

Taback, L., 51
Taylor, B., 402, 413
Taylor, J. C., 293
Teale, K., 52
Teedle, J. B., 52
Terckhov, G. A., 52
Thompson, G. L., 379
Thornton, C., 192, 378
Throndsen, E. C., 52
Tillitt, H. A., 534
Toan, A. B., Jr., 293
Tomin, E. D., 51
Tonge, F. M., 239, 378, 381
Tonik, A. B., 125
Toronto, A. F., 379
Troesch, A., 50
Tunis, C. J., 381

Uhr, L., 382
Ulrich, W., 536
Unger, S. H., 382
Urban, G. H., 51

Van Horn, R. L., 49, 292, 293
VanZoeren, H., 378
Vazsonyi, A., 293
Veasey, L. G., 379
Veitch, E. W., 481*n.*, 494
Venn, J., 444
Von Neuman, J., 6
Vossler, C., 382

Waldo, W. H., 535
Wang, H., 192, 381
Warner, H. R., 379
Warshaw, M. J., 52
Wattenburg, W. H., 238
Wegstein, J. H., 389*n.*
Weinrauch, H., 51
Weiss, E., 157
Weiss, H., 49, 293
Wheeler, D. J., 50
Whitehead, A. N., 435, 493
Wildhack, W. A., 535
Wilkerson, M., 239
Wilkes, M. V., 6, 50
Williams, F. A., 192
Williams, J. D., 330*n.*, 379
Williams, J. J., 157
Williams, S., 6
Wilson, J. B., 378, 459*n.*
Winslow, D. J., 157
Wise, C. S., 534
Wiseman, R. T., 293
Wolf, A. K., 239, 378, 380
Woodger, M., 239
Wooldridge, P. E., 433
Wrigley, C. F., 51
Wrubel, M. H., 50
Wyllys, R. E., 381

Yngve, V., 378
Youle, P. V., 52
Yovits, M. C., 382

SUBJECT INDEX

Page references in **boldface** type indicate definitions

A, COBOL, 263
Absolute accuracy, 425
Absolute value, in ALGOL, 234
Abstraction, 362–371
Accelerating convergence of iterative processes, 389–393
Accounting, 137, 142
Accumulate instruction, Honeywell 800 computer, 540
Accumulator, 32–33, 37–38, 91–93
  addressing, 92
  of digital differential analyzer, 128
Accuracy, 421–429
  absolute, 425
  relative, 424–425
Acoustic delay-line memory, 47
Actual parameter, ALGOL, **229**–232
Actual-parameter list, ALGOL, **229,** 234
Actual-parameter part, ALGOL, **229**
ADD statement, COBOL, **249**–250
Addend, COBOL, **249,** 250, 253
Addend list, COBOL, **249,** 250
Adder, 36–38
  serial, 36–38
  (*See also* Addition)
Adding operator, ALGOL, **212**
Addition, binary, 61–62
    step-by-step example, 36–38
  decimal, 97
  differential, 127
  logical (*see* Logical addition)
  octal, 61–62
  ring, 102
  sexadecimal, 61–62
Addition instructions, types, 100–101
Addition tables, binary, 61
  decimal, 61
  octal, 61
Address, assignment, 185–186
  definition, **31**
  generalized, 143–144
  (*See also* Addressing)
ADDRESS clause, COBOL, 276
Addressing, augmented, 108, 109
  compound, 110–111
    in Honeywell 800 computer, 538, 539
    in IBM 7090 computer, 542–543
Addressing, direct, in Honeywell 800 computer, 538, 540
  indirect, 108–109
    in Honeywell 800 computer, 538, 539, 544
    in IBM 7090 computer, 542–544
  over-, 107–108
  partial, 107
  relative, 110, 112
    in Honeywell 800 computer, 538, 539
    in IBM 7090 computer, 542–543
  variable word, 106–108
Adjective, English and Korean, **301,** 303, 306
ADVANCING clause, COBOL, **285**
AFTER, COBOL, 253, 285
Airline reservation system, 23–25
Aitken's method, 400–401
Algebra (*see* entries under Algebraic; Boolean algebra; Simultaneous linear equations)
Algebraic compiler, 186–191
  flow chart, 188
  writing, 187–189
Algebraic decoder, 186–189
Algebraically oriented language (*see* ALGOL)
ALGOL, 193–239
  anatomy, 204
  automatic translation, 299–300, 311–324
  bibliography, 238–239
  characters, 205–206
  COBOL and, differences, 240–241, 245, 251–252, 254, 256
  declarations (*see* Declarations)
  expressions (*see* Expressions)
  history, 203
  introduction to, 193–202
  purpose, 202–203
  statements (*see* Statement)
  structures, 204
Algorithmic language (*see* ALGOL)
Alphabet, Korean, 195, 196
ALPHABETIC class, COBOL, 264
Alphabetical data, COBOL, standard positioning, 280
ALPHANUMERIC class, COBOL, 264

Alphanumerical data, COBOL, standard positioning, 280
ALTER statement, COBOL, 250–251
Alternative, notation for, ALGOL, 199
COBOL, 241, 285
syntax definition, **306**
Alternative data-names, COBOL, 267–270
Alternative decisions, 325
under conditions of risk, 327–328
sequence, 330–333
Alternative diagnostic probabilities, 341–344
Analogy, hand computation, 28, 29
Analysis, logical, experimental results, 484–486
intelligence reports, 486–487
numerical, 383–434
references, 433–434
syntactical, 295
Anatomy of language, 204
AND, COBOL, 242, 247–248
and, logical, 437
Antecedence solution, 483–484
Applications of digital computers, 1–26
aids to medical diagnosis, 25–26
airline reservation system, 23–25
biomedical, 10–12
data processing, 20–26
machine-tool control, 12–14
man-machine simulation system, 18–21
motion of charged particles (Argus project), 8–10
numerical solutions to equations, 8–12
process control, 12–15
references, 49–52
simulations, 15–21
Approximation, continued-fraction, 394–395
errors in, 422
polynomial (*see* Polynomial approximations)
series, 393–394
successive, 387–393
ARE options, COBOL, 272–273
Arguments, definition, **26–27**
selection, 28–30, 33–34, 39
(*See also* Addresses; Addressing; Memory)
Argus project, 8–10
Arithmetic, floating, 98–100
Arithmetic expression, ALGOL, 208, 209, 211, **213,** 214, 217, 221, 228
simple, 211–**212,** 213
COBOL, **213,** 247–249
Arithmetic instructions, 100–102
Honeywell 800 computer, 540
IBM 7090 computer, 543
UNIVAC 1103A computer, 541
Arithmetic operations, number-system, 61–62
significant figures, 426–429
Arithmetic operator, ALGOL, **206,** 212
COBOL, **242**
Arithmetic units, 28
control, 28–30, 39
Array, ALGOL, 230
Array, in COBOL, 265–267
in solving partial differential equations, 416–421
Array declaration, ALGOL, 226–228, 230
Array-manipulation computer, 149–150
(*See also* Business logistic computer)
Array notation, logistics computer, 143
Arrow, ALGOL, 212
COBOL, 248
Article, definite, English, **301**–303, 306
ASSIGN OBJECT-PROGRAM TO, COBOL, 276
ASSIGN TO, COBOL, 275
Asssignment, COBOL, **275**–276
Assignment list, COBOL, **276**
Assignment statement, ALGOL, **217**–219, 228
automatic translation, 311–324
modified syntax, for efficient translation, 320–321
for translator, 312
Association unit, 369
AT END clause, COBOL, 284–285
Augmented addressing, 108–109
AUTHOR clause, COBOL, 277
Automatic program, 159
compiler routine, address assigner, 185–186
advantages, 180–181
algebraic, 186–191
decoder, algebraic symbols, 186–189
loop symbols, 190–191
recursive, 190–191
translator, 182–185
interpretive routines, data-handling, 172–179
simulational, 168–172
(*See also* Automatic programming)
Automatic programming, 159–192
advanced concepts, 295–298
computing speed and, 179–181
memory allocation, 179
references, 192
Automatic-programming language, 159
ALGOL, 193–239
COBOL, 240–294
introduction to, 193–202
translation, 298–324
references, 378
Automonitor program, 161–163
Auxiliary switches, 104–105, 121
Axiomatic development of Boolean algebra, references, 493–494
Axioms, geometric, 350

B (blank space), COBOL, 263, 281
Babbage's analytic engine, 2
Babel, 299, 300
Basic symbols, Boolean algebra, 448*n.*
Basis, Boolean algebra, 448–449
constrained, 468–469
for dependent elements, 474
designation numbers with respect to, 469–471

Basis, constrained, determining, 468–469
logical, 448–449
symptom-disease complexes, 336, 341
positions of columns, 449
standard, 449
and truth table, 448
Bayes' formula, 342–344
BEFORE, COBOL, 285
BEGINNING-FILE label-record type, COBOL, 273
BEGINNING-TAPE label-record type, COBOL, 273
Best-fit polynomial approximation, 402–404
Bibliography (*see* References)
Binary addition table, 61
Binary arithmetic operations, 61–62
Binary coded decimal, 96–97
Binary codification, 523
Binary-decimal conversion, 59–60
Binary multiplication table, 61
Binary number system, 55, 56
Binary numbers and bits, 35
Binary-octal conversion, 59
Binary point, COBOL, 281
(*See also* Decimal point; Period)
Binary-sexadecimal conversion, 59
Binary words, 35–36
Binomial-coefficient look-up routine, 78
Biochemical logical problems, 484–486
Biomedical research, 10–12
(*See also* Medical diagnosis)
Bit-handling instructions, 102–103, 136
BITS, COBOL, 281
Block, ALGOL, 215–217, 219
grammar, 228
COBOL, 243, 272
of words, 105
BLOCK CONTAINS clause, COBOL, 372
Block diagram, digital computer, 26–31, 33, 136, 138, 140, 150, 152
heuristic program, 354
types of heuristics, 356
Body, procedure, ALGOL, **230**–232
Boolean algebra, 435–494
applications in logical problems, 483–487
as algebra, of propositions, 440
of sets, 442–444
constrained, 471–472
constraints, 467–472
designation numbers, 447–450
equation solution, 482–483
equations characterizing, 440–441
expressions, 439–442
forms of functions, 451–460
historical background, 435–436
logical dependence, 472–474
matrices, 474–487
propositions, 436–438
references, 493–494
representation, hypercube, 446
lattice diagram, 446–447
set, 442–444
searching problem, 499–500
simplification methods, 454–467
Boolean algebra, transformations, 479–482
(*See also* entries under Logical)
Boolean algebraic equations, 482–483
Boolean expression, ALGOL, 208–**211,** 217, 219–220
simple, 209–**210,** 211
COBOL, **247**–248, 251
in logic, 439–442
Boolean function, canonical (normal) forms, 451–452
constrained basis, 469–471
definition, **438**
designation number, 449–450
E, 479–481
F, 479–481
f_s, 479
logical dependence and independence, 472–474
mongrel form, 452–453
relation to Boolean matrices, 479–481
simplest product of sum, 451, 489–490
simplest sum of products, 454–467
simplification, references, 494
(*See also* Boolean algebraic equations)
Boolean lattice diagram, 446–447
Boolean matrices, 474–487
equality, 475
implication, 474–475
logical addition, 474
logical multiplication, 474
transpose, 474
Boolean matrix equations, solutions to, 475–476
all, 476*n.*, 493
unitary, 476–478
Boolean operations, ALGOL, truth-value definitions, 208
Boolean term, ALGOL, **209**–211
Boolean transformations, 479–482
Boolean type, ALGOL, 226, 230
Borrow, 62
Boundary conditions, 333
Bounds, index, ALGOL ***array*** declaration, 226, 227
matrix solution, greatest lower, 476, 477–478
least upper, 475–476, 477–478
lowest, 476, 477–478
Brackets, square, in ALGOL, 207, 226, 227
Brain mechanism, 367
Branch link, list, 297
Break-point, 161
Bugs in program, 160
Business logistic computer, 140–150
basic operations, 142–147
example, building-material purchase, 141–142
keeping bank accounts, 142
spare-parts inventory, 140–141
illustrations of coding, 147–148
instruction format, 142–145
Business-oriented language (*see* COBOL)
Business simulation (industrial dynamics), 15–19
BY, COBOL, 250, 253, 255

Calculus (*see* Boolean algebra; Differential equations; Differentiation; Integration)
Canonical forms of Boolean functions, 451–452
Capacitor memory, 47
Cards, marginal-punch, 504, 532, 533
  peek-a-boo, 504–505
  punched, 105
  searching, 504–506
  sorting, 504–505, 507, 532, 533
Carry, 62
Cell, 31
  (*See also* Memory)
Certainty, decision under, 325–326
Character, ALGOL, 205–206
  automatic-programming language, 194–195
  COBOL, 241–242
  list, 297
Characteristics, COBOL, description, 263–264
  searching, 148–149
    definition, **495**
Chart, prime implicant, 457–459
CHECK PROTECT clause, COBOL, 282
Chemical plant control, 14–16, 132–137
Chinese, 195, 196
Class, 442
  COBOL, 261
  equivalence, 498
CLASS clause, COBOL, **264,** 270
Classes, calculus of (*see* Boolean algebra)
Clause, COBOL, **ADDRESS**, 276
  ADVANCING, **285**
  AT END, 284–295
  AUTHOR, 277
  BLOCK CONTAINS, 272
  CHECK PROTECT, 282
  CLASS, **264,** 270
  data-editing, 280–282
  DATE WRITTEN, 277
  editing, 282
  FILE CONTAINS, 271–272
  FILE CONTROL, 276, 277
  FLOAT DOLLAR SIGN, 282
  FROM, **285**
  JUSTIFIED, 280–281
  MEMORY SIZE, 276–277
  OBJECT-COMPUTER, 276
  OCCURS, 265–267, 271
  PICTURE, 263–264, 270, 281
  POINT LOCATION, 281
  PROGRAM ID, 277
  RECORD CONTAINS, 272
  REDEFINES, 267–268, 271
  REMARKS, 277
  RENAMES, 268–270
  SIZE, 263, 265, 270
  USAGE, **264,** 270
  VALUE, 264–265, 270–271
  VALVE OF, 273–274
  ZERO SUPPRESS, 282
Clock (oscillator), 36
Clock pulse, 36
CLOSE statement, COBOL, **283,** 284
Close-up-space pseudoinstruction, 176
COBOL, 240–294
  ALGOL and, differences, 240–241, 245, 251–252, 254, 256
  automatic translation, 299–300
  character, 241–242
  complete example, 285–289
  data description, detailed, 263–271
  data handling, 277–289
  data organization, over-all, 257–263
  environment description, 275–277
  file description, 271–275
  history, 240–241
  introduction to, 240–246
  operational statements, 247–257, 277
    (*See also* Statement)
  record description, 260–263
  references, 291–294
  words, **242**–244
Code (codification), definition, **498**
Code (program), definition, **7, 41, 53**
Coder, definition, **41**
Codes, decimal, 96–97
  excess-three (decimal-coded binary), 96–97
  multiple, 152–153
  octal-coded binary, 63
  recursive, in ALGOL, 221
  symbolic, Honeywell 800 computer, 540–541
    IBM 7090 computer, 543–544
    instructions for translator, 311
    UNIVAC 1103A computer, 541–542
Codification, binary, 523
  irredundant, 526–532, 534
  nonsubstantive, 521
  redundant, self-correcting, 523–526
  substantive, 521
  superimposed, irredundant, 526–532, 534
Codifying, decoding, 521–522
  definition, **498**
  efficiency, 522–523
  encoding, 521–522
  purposes, 521
Coding, 53–157
  decision, 67–69
  introduction to, 53–54
  minimum access, 96
  principles, 53
  special techniques, 112–118
    input-output instructions, 115–116
    multiway branch, 112–113
    switch sensing, 113–115
    table look-up, 77–78, 116–118
  tactics, 53
  (*See also* Codification; Codifying; Programming)
Coding manual, 63–64
Coding sheet, 41
Collection, empty, 508
Column-adding routine, 78–79
  (*See also* Business logistics computer)
Combined element, definition, **438**
  designation number, 449–450
  (*See also* Boolean function)

Comma, ALGOL, 206, 226, 229, 230
  COBOL, **242**, **249**, **255**, 265, 273, 275, 276, **279**, **283**
Commands, definition, **31**
  (*See also* entries under Instruction)
comment declaration, ALGOL, 233
Common business-oriented language (*see* COBOL)
Comparison, dummy, 90
  logical, 102–103
Comparison instruction, 67, 89, 91, 94, 104, 147
  Honeywell 800 computer, 540
  UNIVAC 1103A computer, 542
  (*See also* Conditional statement; Decision instructions)
Compiler, in COBOL, 248
  recursive, 190–191
Compiler routines, 164, 180–191
  address assigner, 185–186
  advantages, 180–181
  algebraic, 186–191
  decoder of algebraic symbols, 186–189
  flow chart, 184
  kinds, 182
  memory required, 189
  purpose, 182
  translator, 182–185
Complement, Boolean matrix, 474
  logical, 103
  (*See also* Inversion; Negation; *not*)
Complex, symptom-disease, 337
  probabilities, 343
Component, key, translator, 307, 310
  syntax definition, **306**
Composition, musical, by computer, 371–375
Compound addressing (*see* Addressing)
Compound ***if*** statements, ALGOL, 219
Compound statements, ALGOL, 215–217, 219
Computation analogy, 28, 29
Computational-array drum, 143, 149, 150
COMPUTATIONAL usage, COBOL, 264
COMPUTE statement, COBOL, **249**
Computer, analog, 7, 18
  applications, 1–26
    references, 49–52
  array-manipulation, 149–150
  block diagrams, 26–31, 33, 136, 138, 140, 150, 152
  business logistics, 140–150
  decimal, 96–98
  digital, 7
  digital differential analyzer, 127–132
  evolution, 3
  functional description, 31–35
  general-purpose, 7
  history, 2–6
  information-retrieval, 137–140, 149–150
  large-scale data processor, 150–154
    references, 157
  operating modes, 121
  operations performed by, 26–27
  (*See also* entries under Instruction)
Computer, parallel, 38
  real-time control, 132–137
  searching, 137–140, 149–150
  serial, 38
  special-purpose (*see* Special-purpose computers)
  speed, 28, 36, 43, 47, 179–181
  systems-control, 132–137
  terminology, 31–32
  types, 7
Computer name, COBOL, 276
Computer words, 31–32, 35–36, 40–41
Computing unit, 27–28
  of digit-differential analyzer, 128–130
Concepts, abstract, 363–367
    references, 381–382
  generation, 375
Concurrent input-output, 105, 151, 153–154
Condition, tally, ALGOL, **221**–223
Condition description, COBOL, **265**
Condition-name, COBOL, **243**, 246, 247, 265
  value, 264–265
    (*See also* **VALUE** clause)
Condition-value entry, COBOL, **265**
Condition-value list, COBOL, **265**
Condition-variable, COBOL, **245**–247, 264–265
  value, 265
Conditional forms, ALGOL, 219–220
Conditional jump, 93, 94, 104
Conditional probabilities, 341
Conditional statement, ALGOL, **219**–221
  COBOL, **251**–253
  differences, 251–252
Configuration section, COBOL, **276**–277
Conjunction, English, 444
  logical, 437, 438
Conjunctive normal form, 452
Connection unit, 369
Connections, heuristic programming, 358, 366
Connector, 69
  variable, 81
    in ALGOL, 227–228
Consequence solution, 483–484
Consequences of mathematical results, 324
Constant numbers, 66
Constants, in COBOL, 270–271, 274
Constrained basis, for dependent elements, 473
  designation numbers, 469–471
  determination, 468–469
Constrained Boolean algebra, 471–472
Constraints, 467–472
  in medical knowledge, 338–339
  in searching problem, 499
  simplest sum of products under, 470–471
Continued fractions, 394–395
Control, digital, 7
  applications, 1–26
    chemical plant, 14–16, 132–137

Control, digital, applications, inventory, 20–23
machine-tool, 12–14
process, 7, 12–15, 132–137
references, 157
Control generator, 29–30
Control panel, 118–122
buttons and switches, 120–121
lights and signals, 120
purpose, 118–120
Control pulses, 39
Control signals, 39
Control unit, 28–30, 39
Controlled variable, ALGOL, 225
Convergent, continued-fraction, 394
Conversion, number, 54–60
from binary to octal or sexadecimal, 59
of fractional part, from decimal, 58–59
to decimal, 60
of integers, from decimal, 57–58
to decimal, 60
Conversion table, 56
Cookbook language, 200–202
Core memory, 45, 47
Cores, magnetic, 45, 48
CORRESPONDING, MOVE, COBOL, 279
Counterpoint, 372
Counters in digital differential analyzers, 128
Creativity, 363, 371–375
Criteria of progress, heuristic programming, 359
Current-address counter (*see* Current address register)
Current-address register, 33–34, 89
Curve fitting, 14
(*See also* Polynomial approximation)

Data, COBOL, 244–245
positioning, 280–281
definition, **31**
end word for, 176, 177
file, COBOL, **271**–272
organization, bookshelf, 176
for searching, 173
threaded list, 177
Data declarations, cookbook language, 201, 202
Data description, COBOL, 261, 265, 268, 277–278
detailed, 263–271
in **MOVE** statement, 279
special-purpose, 264–270
working storage, COBOL, 270–271
Data division, COBOL, 244–246, 260
Data-editing clauses, COBOL, 280–282
Data handling, COBOL, 277–289
Data-handling interpretive routines, 172–179
Data-handling statements, COBOL, 278–279
Data-item, COBOL, 242, 246
elementary, 262–264
positioning, 280–281
Data-item description, COBOL, **261**
Data-name, COBOL, **242**–244, 247, 249–251, 253, 255, 261, 264, 268, 269, 272, 274, 279, 285
alternative, 267–270
partial, **279**
value, 242–256
initial, 270
Data-name in constants, COBOL, 274
Data-name in data-record, COBOL, 274
Data-name in label record, COBOL, 274
Data processing, applications, 20–26
techniques, 383–536
(*See also* Business logistic computer; Data handling; Information retrieval; Searching)
Data-processing language (*see* COBOL)
Data-record name, COBOL, 271, **272**
Data (of program), COBOL, 277
DATE WRITTEN clause, COBOL, 277
Debugging, 160–164
Decimal arithmetic, 61, 96–97
Decimal-coded binary, 96–97
Decimal coding system, 96–98
Decimal conversion, 55–60
Decimal fraction, ALGOL or COBOL, 198–**199**
Decimal number, ALGOL or COBOL, 198–**199**
Decimal point, ALGOL, **198**
COBOL, **198**, 264, 281
(*See also* Period)
Decision instructions, 103–105, 112
coding with, 67–69
(*See also* Comparison instruction; Conditional statement)
Decision theory, modern mathematical, 324–333
references, 378–379
Decisions, management, 17
Declaration, ALGOL, ***array***, 226–228, 230
comment, 233
procedure, **230**–234
for function, 234–235
relation to procedure statement, 231–232
switch, 214, **227**–228, 230
type, **226**–228
COBOL (**DEFINE** statement), **256**–257
Declarations, ALGOL, 225–228, 230–234
labeling, 216
cookbook language, 201, 202
need for, 225–226, 228–229
Decoder, algebraic symbol, 186–189
instruction, 28–30, 33, 39
Decoding and encoding, 521–522
Deductive inference, 295–296, 324–333
in abstraction, 363–365
in abstraction formulation, 367–369
example in medical diagnosis, 334–348
DEFINE statement, COBOL, **256**–257
Definite article, English, **301**–303, 306
Definition of a language, **299**
Definitions, recursive, **198**
syntactical, **194, 197–199**
symbols, **199**

Delimiter, ALGOL, **206**
COBOL, **242**
De Morgan's rules, 440
Dependence, logical, 473–474
Dependent variable, ALGOL, 229, 231, 234
COBOL, 255–256
DEPENDING ON, COBOL, 251
Derivation of fundamental matrix formulas, 479–482
Description, COBOL, condition, **265**
data, 261, 265, 268, 277–278
detailed, 263–271
in **MOVE** statement, 279
special-purpose, 264–270
working storage, 270–271
data-item, **261**
environment, 246, 275–277
file, 246, **271**–275
identification, 246
label-record, 271–275
record, 246, 260–**261**, 262–263
Designation numbers, 447–450
basis, 448–449
to Boolean function, 451–467, 470–471
of Boolean functions, 449–450
Boolean matrices and, 479–481
of combined elements, 449–450
under constrained basis, 469–471
Designational expression, ALGOL, 208, 214–**215**, 218, 227
simple, **215**
Designator, ALGOL, function, 207–**208**, 209, 212, 229, 230, 234
switch, **227**–228
Desk-calculator analogy, 28
Destination, COBOL, 251
Devanagari, 195, 196
Diagnosis (*see* Debugging; Medical diagnosis)
Diagnostic probability, 343
alternative, 341–344
Diagram (*see* Block diagram; Flow charts)
Diagram routine, heuristic programming, 354–355
Difference equations, 416–421
Difference quotients, 415–417
Differences, heuristic programming, abstract-concept, 365–366
trigonometric, 355–359
polynomial, 395–397
Differential additions, 127
Differential equations, ordinary, 410–414
partial, 414–421
Differentiation, by definition, 407
by polynomial approximation, 408–409
by undetermined coefficients, 410
Digit, ALGOL or COBOL, **198**, 199, **206**–208
COBOL, **241**
Digital computer (*see* Computer)
Digital differential analyzer, 127–132
computing units, 128–130
concept, 127, 129
programming, 130–132
purpose, 127
Digital differential analyzer, references, 156–157
Digital switching circuits (gates), 36
Diode memory, 47
Direct memory addressing, in Honeywell 800 computer, 537, 538
in IBM 7090 computer, 543
Discussion (of program), COBOL, 277
Disease complex, 337
probability, 343
Disease profile, 338
Disjunction, 437, 438
Disjunctive normal form, 451
DISPLAY usage, COBOL, 264
DIVIDE statement, COBOL, **250**
DIVIDED BY, COBOL, 242, 248
Division, arithmetic, in digital differential analyzer, 131–132
(*See also* Division instructions)
COBOL, data, 244–246, 260
environment, 246*n*., 277
(*See also* Environment description)
identification, 246*n*., 277
(*See also* Identification description)
procedure, 244, 246
Division instructions, types, 102
do, ALGOL, 221–225
Dollar sign, COBOL, 242, 263, 282
"Don't-care" conditions (*see* Constraints)
Drum, magnetic, 45, 46, 143
type-face, 44
Drum memory, array of numbers stored, 143
computational-array, 143–150
searching, 149, 150
Dummy comparisons, 90
Dummy instruction, 65
Dynamic program debugging, 160–163
automonitor, 161–163
break-point, 161
Dynamic programming, 330–333
in medical diagnosis, 344–348

E function, 479–481
E matrix, 480–481
Editing clauses, COBOL, 282
Editing symbol, COBOL, **242**
EDSAC, 6
EDVAC, 6
Efficiency, of codifying, 522–523
of syntactical and semantic description, 324
Element, for, ALGOL, 221
set, 442
Elementary data-item, COBOL, 262–264
positioning, 280–281
Elementary elements, 438
independent and dependent, 467–468
Elementary product, 451
Elementary sum, 452
ELSE, COBOL, 251
Else fragment, COBOL, **251**–252
Else statement, COBOL, **251**–252
Empty, ALGOL, 229, 230
COBOL, 251, 253–254

Encoding, 521–522
End-of-data word, 176, 177
ENDING-FILE label-record type, COBOL, 273
ENDING-TAPE label-record type, COBOL, 273
English, analogy with automatic-programming languages, 194, 197, 200, 202, 225–226, 240, 248, 253
  logical translation, 444–446
    references, 494
  semantics, 200
  translating into Korean, 301–306
ENIAC, 2
Entrance, subroutine, 81
Environment, COBOL, **246**
Environment description, COBOL, 246, 275–277
Environment division, COBOL, 246*n.*, 277
Environmental factors affecting diagnosis, 343–344
Enzyme reactions, analyzing, 484–486
Equality, of Boolean matrices, 475
  logical, 102–103
  (*See also* Equivalence)
EQUALS, COBOL, 249
Equations, algebraic, 387–393
    Boolean, 482–483
      proving, 440–441
  difference, 416–421
  linear, simultaneous, 384–387
  numerical solutions as computer applications, 8–12
  ordinary differential, 410–414
  partial differential, 414–421
  transcendental, 387–393
Equivalence, Boolean, 439
Equivalence relation, 498
Error, curves for, in function evaluation, 399, 400, 403, 404
  experimental, 422
  pitfalls from 428–429
  relative, 424–426
  sources, 421–423
  truncated, 421–422
  (*See also* Accuracy)
Error detecting and correcting codes, 523–526
  references, 535–536
Error propagation of polynomial differences, 397
Essential prime implicants, 458
  finding, 461–467
Etiological factors affecting diagnosis, 343–344
Evolution of digital computer, 3
Excess-three code (decimal-coded binary), 96–97
Exclusive *or*, 445
Execution of instruction, 26–27
Exit, COBOL loop, 284–285
  subroutine, 81
Expansion, of function (*see* Functions, evaluation)
  of numbers in powers of radix, 54
Experimental errors, 422
Exponent, floating-point, 99
EXPONENTIATED BY, COBOL, 248
Exponentiation, ALGOL, 212
  COBOL, 248
Expression, ALGOL, 229
  arithmetic, ALGOL, 203–205, 208, 209, 211–**213**, 214, 217, 221, 228
    simple, 211–**212**, 213
      automatic translation, 311–324
    COBOL, **213**, 247–249
  Boolean, ALGOL, 208–**211**, 217, 219–220
    simple, 209–**210**, 211
    COBOL, **247**–248, 251
  designational, ALGOL, 208, 214–**215**, 218, 227
    simple, **215**
Expressions, ALGOL, 206–215
  automatic-programming language, 197
  Boolean, in logic, 439–442
  pictorial, in abstraction, 364–367
Extract instruction, 102–103, 136
  in use, 133–135, 167, 171

F, COBOL, 241
  (*See also* Logical value)
F function, 479–481
F matrix, 480–481
Factor, ALGOL, **212**
False (*see* Logical value)
False drop, probability, 529–532, 534
FD, COBOL, 271
Field, computer-memory, COBOL, 279–281
File, COBOL, **246**, 257–260, 271
    opening and closing, 283
    storing, 257–260
FILE CONTAINS clause, COBOL, 271–272
FILE CONTROL clause, COBOL, 276, 277
File data, COBOL, **271**–272
File description, COBOL, 246, **271**–275
File-handling statements, COBOL, 282–285
File-name, COBOL, 271, 275, 283, 284
File-name list, COBOL, **283**
File-name options, COBOL, **283**
File-name options list, COBOL, **283**
File organization, COBOL, **271**–272
FLOAT DOLLAR SIGN clause, COBOL, 282
Floating point, 98–100
  simulation routine, 168–172
Flow charting, 69–70
  symbols, 69, 74
Flow charts, accelerating convergence of iterative processes, 392
  algebraic-compiler, 188, 190
  COBOL conditional statements, 252
  compiler routine, 184
  computation, of radiation intensity, 83
    of $\sqrt{N}$, 75
  diagnostic-testing iteration, 340
  dynamic diagnostic-treatment cycle, 348
  evaluation of single function, 190
  forming COBOL loops, 255
  generalized loop, 74

Flow charts, input-output routine, 115
  instruction modification, 78
  interpretive routine, 166
    sequencing, 170
  loops within loops, 80
  merging of ordered lists, 516
  missile trajectory, 73
  ordering by merging, 518
  real-time operating program, 135
  processing COBOL records, 260, 284
  recording common numbers on two ordered lists, 175
  setting up subroutines, 82
  solution, of simultaneous linear equations, 387
    of $\sin x = 2 - 2x$, 390
  split-tree method of table look-up, 117
  table look-up routine, 77
  translator, detailed, 308–309
    gross, 307
    use, 301
For element, ALGOL, 221
For list, ALGOL, **221**–223
for statement, ALGOL, 219, **221**–225
  COBOL (**PERFORM** statement), **253**–255
Formal parameter, ALGOL, **230**–232
Formal-parameter list, ALGOL, **230**
Formal-parameter part, ALGOL, **230**
Format, in COBOL, 244–245, **256**
    defining new, 256
  instruction (*see* Instruction format)
  label-record, COBOL, 274
    (*See also* Label-record description)
  list-processing language, 298
  one-address, simulated, 169
  output, COBOL, 280–282
  record, COBOL, 260–263
    (*See also* Record description)
  word, 39, 62–66, 88, 90, 93, 95–96
    Honeywell 800 computer, 538
    IBM computer, 542
    UNIVAC 1103A computer, 541
Forms, conditional, ALGOL, 219–220
  grammatical, 194, 200–202
Formulas, well-formed, in geometry, 350
    in heuristic programming, 357
Formulation of abstractions, 367–371
FORTRAN (*see* ALGOL)
Fraction, decimal, ALGOL or COBOL, 198–**199**
FROM, COBOL, 249, 250, 253, 285
FROM clause, COBOL, **285**
From list, COBOL, **253**–254
Foundations of Boolean algebra, references, 493–494
Four-address format, 39, 62–65
Function designator, ALGOL, 207–**208**, 209, 212, 229, 230, 234
Function subroutine, 164–165
  reused, 180, 181
Functional calculus of symbolic logic, 436
Functional description of computer, 31–35
Functions, Boolean (*see* Boolean function)
  evaluation, 393–404, 409–410
Functions, evaluation, best-fit polynomial approximations, 402–404
    expansion, continued-fraction approximation, 394–395
    series of approximations, 393–394
  flow chart, 190
  interpolation, 398–401
    Aitken's method, 400–401
  polynomial approximations, 393, 398–401
  undetermined coefficients, 409–410
  given in logical problems, 483
  logical, ALGOL, 210
  medical knowledge, 337
  procedure statements, ALGOL, 234–235
  (*See also* Numerical analysis; Subroutine)
Fundamental formulas, Boolean transformation, 481–482

Game theory, 328–330
Gates, 36
  digital differential analyzer, 128–130
General problem-solving program, 355–362
General-purpose computer, 7
Generalized address, 143–144
Geometry, plane, 349–354
GIVING, COBOL, 249–250
go to statement, ALGOL, 214, **218**–219
    with conditional statement, 220–221
    in ***for*** statement, 225
    in ***switch*** declaration, 227–228
GO TO statement, COBOL, 250–**251**, 253
Grammar, ALGOL, of blocks, 228
    statement, assignment, 218
      conditional, 221
      for, 225
      go to, 219
      procedure, 229
Grammatical forms of language, 194, 200–202
Greatest lower bound of matrix solutions, 476–478

Hardware, in COBOL, 275–277
Hardware list, COBOL, **275**, 276
Hardware-name, COBOL, 275–276
Harmonic rules, 372–373
Harris-Elbourn method of simplification, 460–467
HASHED option, COBOL, 273–274
Heading, procedure, ALGOL, **230**–232
Health-computing system, 25–26
Heuristic programming, 348–355
  comparison with translator, 355
  references, 379–381
  technique, 351–354
Heuristic routine, 354–355
Heuristics, 349, 351
  semantic, 352, 354
  source in geometry, 350, 351
  syntactic, 352, 353
  syntactic-symmetry, 353
  types, 355–357
Hexadecimal (*see* Sexadecimal)

High-speed printer, 43–45
History, ALGOL, 203
  Boolean algebra, 435–436
  COBOL, 240–241
  computers, 2–6
Hodgkins-Huxley equations, 11
Honeywell 800 computer, 537–541
House, idealized concept, 366
Hypercube representation of Boolean algebra, 446
Hypotheses, good, choosing, 349
  in logical problems, 483
Hyphen, COBOL, 242

IBM 7090 computer, 542–544
Identification, abstract, 363–367
  program, COBOL, **277**
Identification description, COBOL, 246
Identification division, COBOL, 246*n.*, 277
Identifier, ALGOL, **207***n.*
    procedure, 207–**208, 229,** 230
  automatic-programming language, 197
IF, COBOL, 251
If clause, ALGOL, **211,** 213–215, 219
If fragment, COBOL, **251**–252
if statement, ALGOL, **219**
  COBOL, **251**–252
Illiac, 375*n.*
Implicant, definition, **454**
  (*See also* Prime implicant)
Implication, ALGOL, **210**
  Boolean matrices, 474–475
  logical, 439
In-out file-name list, COBOL, **283**
In-out selector, 30, 153
  (*See also* Interlocks)
INCLUDE statement, COBOL, 254–**255,** 256
Inclusive *or*, 445
Independence, logical, 472–473
Independent symptoms, 344
Independent variable, ALGOL, **208,** 229–231, 235
  COBOL, 255–256
Independent-variable list, ALGOL, **208,** 234
Index, ALGOL, ***array*** declaration, 226–227
    for statement, 223, 225
  translator subroutine, 311
Index-manipulation instructions, IBM 7090 computer, 543–544
Index registers, 110
  Honeywell 800 computer, 537–540, 544
  IBM 7090 computer, 542–544
  in list processing, 298
  use in push-down lists, 178–179
Indexing (*see* Tabledex)
Indirect addressing (*see* Addressing)
Inductive inference, 295–296, 348–355
  in abstraction, 365–367
  example of general problem solver, 355–362
Industrial dynamics, 15–19
Inequality, logical, 102
Inference (*see* Deductive inference; Inductive inference)
Information retrieval, 8
  references, 534–535
  (*See also* Data processing)
Ingredients, of formal mathematical system, 350
  of a language, 194–197
  of medical diagnosis, 335
INPUT, COBOL, 283
Input-file area, COBOL, filling, 283–284
    transfer from, 278–279
Input files, COBOL, 257–260, 283
Input numbers, 6–7, 26
Input-output, concurrent, 105, 151, 153–154
Input-output equipment, in COBOL, 275–277
Input-output methods, 30, 41–44, 45
  speeds, 43
Input-output problems, large-scale data processor, 150–151
Input-output section, COBOL, **276**–277
Inscriber, 41
Instruction decoder, 29–30, 33, 39
Instruction execution, 26–27
Instruction format, business logistic computer, 142–145
  four-address, 39, 62–65
  list-processing language, 298
  one-address, 93–94
    IBM 7090 computer, 542
    simulated, 169
  retrieval computer, 138
  three-address, 88
    Honeywell 800 computer, 538
  two-address, 90
    UNIVAC 1103A computer, 541
Instruction modification, 32, 77–79, 111
Instruction register, 33, 39
Instruction sequences, 62–67
Instruction types, 98–106, 112
  arithmetic, 100–102
  bit-handling, 102–103, 136
  close-up-space, 176
  comparison of words, 67, 104, 147
  decision, 67–69, 103–105
    (*See also* Conditional statement)
  extraction, 103–136
    using, 167, 171
  Honeywell 800 computer, 540
  IBM 7090 computer, 543–544
  input-output, 105–106
  jump, 104, 112
  list-processing language, 297–298
  logical, 102–103, 136
  make-space, 176
  one-address, simulated, 169
  read and write, 105–106
  shift, 103
  switch sensing, 104
  transfer, 167, 170
  translator target language, 311
  UNIVAC 1103A computer, 541–542
Integer, ALGOL, 227
  COBOL, 263–265, 271, 272, 281, 285
  unsigned, ALGOL or COBOL, 198–**199,** 215

integer type, ALGOL, 226, 230
Integrating general- and special-purpose computers, 139–140, 151–154
Integration, by definition, 404–405
  on logistics computer, 148
  by polynomial approximations, 406–407
  by Simpson's rule, 406–407, 409–410
  by trapezoid rule, 406
    in ALGOL, 233
  by undetermined coefficients, 409–410
Intelligence achievement, mathematics in, 325
  programming in, 295–382
Intelligence reports, logical analysis, 486–487
Interlocks, 150–154
International Algebraic Language (*see* ALGOL)
Interpolation, 398–401
Interpretation of word, 32, 35
Interpretive routines, 164–179
  complications, 179–180
  data-handling, 172–179
  mathematical, 164–168
  simulational, 168–172
Intersection of sets, 443
INTO, COBOL, 250
Intuition, 349
Inventory control, 20–23
  (*See also* Business logistic computer)
Inversion, logical, 450
  (*See also* Complement)
Irredundant codifying, 526–532, 534
  definition, 522
Irrelevant symptoms, 369
IS EQUAL TO, COBOL, 242, 247–248
IS GREATER THAN, COBOL, 242, 247–248
IS LESS THAN, COBOL, 242, 247–248
IS options, COBOL, 272, 273
Item, 148–149, 495
Item-list searching, 500
Iteration, 74
  COBOL, 254
  diagnosis-testing, 335, 339–340
  diagnosis-treatment, 344–348
  difference equation, 418–420
  in dynamic programming, 333
  in successive approximations, 387–393
  (*See also* Recursion codes)
Iterative process, accelerating, 389–393

Jacquard loom, 2
Jump, 80*n.*
  negative, 104
Jump instructions, 104, 112
  ALGOL, 214
    (*See also* ***go to*** statement, ALGOL)
  conditional, 93, 94, 104
    (*See also* Comparison instruction; Conditional statement; Decision instructions)
  IBM 7090 computer, 543–544
  unconditional, 91, 93
  UNIVAC 1103A computer, 542
Jump table, 113, 114, 133
JUSTIFIED clause, COBOL, 280–281

Key component, translator, 307, 310
Key symbol, translator, 307, 310
Korean, translating English into, 301–306
Korean alphabet, 195, 196

Label, ALGOL, **215**–217, 219, 228
    if statement, 219–220
    in ***switch*** declaration, 227
  automatic-programming language, 197, 214
  of block of data, 173
  COBOL, **246,** 251, 253, 255
Label list, ALGOL (switch list), **227**
  COBOL, **251**
Label record, COBOL, 272–275
Label-record description, COBOL, 271–275
Label-record type, COBOL, 271–**273,** 274
Label-record-type list, COBOL, **273**–274
Labeling words of lists, 173
Language, anatomy, 204
  automatic-programming (*see* Automatic-programming language)
  cookbook, 200–202
  definition, **299**
  ingredients of, 194–197
  list-processing, 296–298
  object, 299*n.*
  source, 299
  target, 299
    instructions for translator, 311
  (*See also* ALGOL; COBOL)
Laplace's equation, 416, 418–421
Lattice array in solving partial differential equations, 416–421
Learning, concept, 369–371
  heuristic programming, 362
Least upper bound of matrix solution, 475–478
Left part, ALGOL, **217**
Left part add, COBOL, **249**
Left part compute, COBOL, **249**
Left part perf, COBOL, **253**
Left part sub, COBOL, **250**
Length of word, 35
Letter, ALGOL, **206**–208
  COBOL, **241**
Level number, COBOL, 261–263, 265, 268
  syntax definition, for translator, 304, 306, 310–311
Level **66**, COBOL, 269
Level **77**, COBOL, 270
Level **88**, COBOL, 265, 271
Library of subroutines, 84–85
Line, syntax definition, 306
Linear algebraic equations, simultaneous, 384–387
Linear Boolean equations, 482–483
Linear ordering, 498
Linear programming, 325–326
LINES, COBOL, 285
Link, list, 297

List, ALGOL, actual-parameter, **229,** 234
for, **221**–223
formal-parameter, **230**
independent-variable, **208**
label (switch), **227**
variable, **226, 230**
characteristic, 495
COBOL, addressed, **249,** 250
assignment, **276**
condition-value, **265**
file-name, in-out, **283**
file-name-options, **283**
from, **253**–254
hardward, **275,** 276
label, **251**
label-record type, **273**–274
memory-segment, **276**
partial-data-name, 279
receiving-area, **279**
receiving-field, **279**
record-name, **272**
replacement, **255**
end of, word for, 176, 177
item, 495
ordering, 498, 510–521
population-based, 503
push-down, 178–179, 296–297
translator, 311*n.*, 313
searching, 500–501
source code, 306
substantive-based, 503
threaded, 177
push-down lists in, 179
List processing, 296–298
references, 377–378
Literal, COBOL, **242**–243, 246, 247, 249, 265, 276
condition-name, 243
English, 304
Local, ALGOL, 216–217, 219, 228
LOCK option, COBOL, 283
Logic (*see* Boolean algebra; Symbolic logic)
Logical addition, 102, 437, 438, 449
Boolean matrix, 474
Logical basis (*see* Basis)
Logical comparison, 102–103
Logical-control computer, 132–137
Logical dependence and independence, 472–474
Logical equality, 102–103
Logical inequality, 102
Logical instructions, 102–103, 136
Honeywell 800 computer, 540
IBM 7090 computer, 544
UNIVAC 1103A computer, 541
Logical multiplication, 102, 437, 438, 449–450
Boolean matrices, 474
Logical operation, COBOL, **247**
(*See also* Logical operator)
Logical operations, 437–438, 441–442
(*See also* Boolean operations)
Logical operator, ALGOL, **206**
COBOL, **242**
(*See also* Logical operation)
Logical problems, 483–484
Logical translation of English, 444–446
references, 494
Logical value, ALGOL, **206,** 209
in ALGOL Boolean expression, 208–211
COBOL, **241,** 247
Logistic computer (*see* Business logistic computer)
Look-up routine, binomial-coefficient, 78
Loops, in ALGOL, 221
in COBOL, flow chart, 255
within loops, 79–80
in ALGOL, 225
in COBOL, 254
(*See also* Recursion codes; Recursive compiler)
Lowest solution, Boolean matrix equations, 476–478

Magnetic tape, 41–43, 105
in COBOL, 244–246, 257–260, 262, 271–274
rewinding, 283
Major multiplication, 101
Make-space pseudoinstruction, 176
Man-machine simulation, 18–21
Management decisions, 17
Mantissa, floating-point word, 99
Manual searching methods, 504–506
Marginal-punch cards, 504, 532, 533
Masked instructions, Honeywell 800 computer, 540
Mathematical decision theory, 324–333
references, 378–379
Mathematical interpretive routine, 164–168
Mathematical optimization, 324
(*See also* Deductive inference)
Mathematical system, formal, 350
Matrix, in ALGOL, 226–227
Boolean (*see* Boolean matrices; Boolean matrix equations)
in COBOL, 265, 267
permutation, 476
pseudopermutation (unitary matrix), 476–478
transformation, Boolean, 479–482
Matrix equation for continued fraction, 394–395
(*See also* Boolean matrix equations)
Matrix multiplication, in ALGOL, 233
Medical diagnosis, aids to, 25–26
references, 379
factors affecting, 343
formulation, 334–335
ingredients, 335
programming for, 334–348
reasoning in, 334–335
Medical knowledge, 335–338
probabilities in, 341–344
Melodic rules, 372, 373
Memory, 27–28, 31, 43–48
high-speed, in COBOL, 259–260, 268, 270, 272, 274, 276, 283
sharing, 179
reading from, 47–48

Memory, required for compiling routine, 189
speed of, 47
Memory access time, definition, **47**
Memory address, definition, **31**
Memory segment, COBOL, **276**
Memory-segment list, COBOL, **276**
MEMORY SIZE clause, COBOL, 276–277
Memory space, sharing, 179
Memory systems, 44–48
Merging, ordering by, 515–521
Metalanguage, 199, 299, 301
Meta-metalanguage, 199
Minimization, linear-programming, 326
Minimum, maximizing, 329
Minimum-access coding, 96
Minor multiplication, 101
MINUS, COBOL, 242–248
Mixed strategy, 329
Models, mathematical (*see* Simulation)
Modes of operation, 121
Modification, automatic-programming language, 299
instruction, 32, 77–79, 111
Mongrel form, 452–453
MOVE CORRESPONDING statement, COBOL, **279**
MOVE statement, COBOL, 278–**279,** 280–282, 285
Mult-factor, ALGOL, **212**
Mult operators, ALGOL, **212**
Multiple codes, 152–153
Multiplication, and division, 61–62
floating, 99–101
logical, 102, 437, 438, 449. 450
major, 101
matrix, in ALGOL, 233
Boolean, 474
minor, 101
Multiplication instructions, types, 101
Multiplication tables, 61
MULTIPLY statement, COBOL, **250**
Multiway branch, coding, 112–113
Music composition by computer, 371–375

n-phrase, English and Korean, **301**–303, **306**
Name (of program author), COBOL, 277
Napier's "bones," 2
Neg-term, ALGOL, **210**
Negation, 437, 438
(*See also* Complement)
Negative jump, 104
(*See also* Conditional jump)
Nets, self-organizing, 369
Neural-net simulation, 369
Neuron, 369–370
9 (numerical character), COBOL, 263
NO REWIND option, COBOL, 283
Nonconditional statement, COBOL, 284
Noncontiguous working-storage, COBOL, 270
Nonlocal, ALGOL, 217, 228
Normalizing, 99
Normalizing instruction, UNIVAC 1103A computer, 542
NOT, COBOL, 242, 247–248
not, logical, 437, 438
Notation, conditional probability, 341
translator, 303, 312
(*See also* Symbols)
Noun, English and Korean, **301**–303, 306
Number, ALGOL, 206
ALGOL or COBOL, 198–**199**
automatic-programming language, 197
decimal, ALGOL or COBOL, 198–**199**
level, COBOL, 261–263, 265, 268
Number systems, 54–62
Numbers, input-output, 6–7, 26, 40
random, in musical creativity, 329, 372–375 types in computers, 66
NUMERIC class, COBOL, 264
Numerical analysis, 383–434
references, 433–434
Numerical approximations, errors in, 422
Numerical data, COBOL, standard positioning, 280–281
Numerical solutions to equations, 8–12
Numerical value, 211, 213–214

Object, heuristic programming, in abstraction, 365
OBJECT-COMPUTER clause, COBOL, 276
Object language, 299*n.*
Object phrase, English and Korean, **301**
OBJECT-PROGRAM assignment, COBOL, 276
OCCURS clause, COBOL, 265–267, 271
Octal number system, 55, 56
addition tables, 61
conversion, 55–60
multiplication tables, 61
OMITTED label-record description option, COBOL, 273
One-address format, 93–94
One-address instruction system, 93–95
example, 542–544
simulating in two-address, 168–172
One-plus-one instruction address system, 95–96
OPEN statement, COBOL, **283,** 284
Operand, relational, COBOL, **247**
Operation, ALGOL, Boolean, truth-value definition, 208
COBOL, logical, **247**
relational, **247,** 248
propositional, 437–438
(*See also* Operator)
Operational characteristics, COBOL, description, 263–264
Operator, ALGOL, adding, **212**
arithmetic, **206,** 212
logical, **206**
mult, **212**
relational, **206,** 209
COBOL, arithmetic, **242**
logical, **242**
relational, **242**
heuristic programming, in abstraction, 365
transformation, 352

Operator, heuristic programming, in trigonometry, 356–357, 358–359
(*See also* Operation)
Optimality principle, 333
Optimization, mathematical, 324
(*See also* Deductive inference)
Option-value, COBOL, **245**, 264–265
OR, COBOL, 242, 247–248
or, inclusive and exclusive, 445
logical, 437
Ordering, 510–521
definition, **498**
methods, comparison, 520–521
maximum in-pass, 510–511
minimum in-pass, 510–511, 512
merging, 515–521
radix coefficients, 512–515
transposition, 511–513
population-based, 503
references, 534
substantive-based, 503
Organization, COBOL, file, **271**–272
over-all, 244–246
data, bookshelf, 176
COBOL, 257–263, 271
for searching, 173
threaded list, 177
Oscillators, clock, 36
Out-scriber, 43
OUTPUT, COBOL, 283
Output devices, 30, 41–45
Output file, COBOL, 257–260, 283
Output-file area, COBOL, emptying, 283–284
transfer to, 278–279
Output format, COBOL, 280–282
Output numbers, 6–7, 26
Overflow, 100
in digital differential analyzer, 128, 129

Paper tape, punched, 30, 40–41, 105
Paragraph, COBOL, 243, 246, 255
GO TO statement, 251
Parameter, ALGOL, actual, **229**–232
formal, **230**–232
Parentheses, 187
in ALGOL, 207–209, 212, 215, 229–231
translating, 316, 317
omitting, 209, 212
Partial addressing, 107
Partial data-name, COBOL, **279**
Partial-data-name list, COBOL, 279
Partial differential equations, 414–421
Pattern recognition, 363–367, 369–371
references, 381–382
Payoff, game theory, 329
Peek-a-boo cards, 504–505
Perceptron, 369
PERFORM statement, COBOL, **253**–255
Period, COBOL, 242, 246, 251, 263
(*See also* Decimal point)
Permanent conditions in searching, 508
Permanent numbers, 66
Permutation matrix (unitary matrix), 476–478
Phase-selector switch, 121
Phases, operational, 34
Philosophy of programming, 54
Physical characteristics, COBOL, description, 263–264
Pictograph, 195, 196
Pictorial expressions in abstraction, 364–367
PICTURE clause, COBOL, 263–264, 270, 281
Picture-reading machine, 14
Pitfalls in using computer, 428–429
PLACES, COBOL, 281
Plane geometry, 349–351, 352–354
Planning, heuristic programming, 362
Players, game theory, 328
PLUS, COBOL, 242, 248
Point (*see* Decimal point)
POINT LOCATION clause, COBOL, 281
Polynomial, differences, 395–397
Polynomial approximation, 393, 398–401
best fit, 402–404
differentiation by, 408–409
integration by, 406–407
series, 393–394
solving differential equations by, 411–413
Polynomial interpolation, 398–401
Popping up, 178
Population of patients, 343
Population-based ordering, 503
Position, elementary data-item, COBOL, 280–281
Post-mortem program debugging, 160–161, 163–164
Power positions, definition, **461**
Power series approximation, 393–394
Precedence rules, ALGOL, arithmetic, 212
relational and logical, 209
COBOL, arithmetic, 248
logical, 248
Premises in logical problems, 483
Prime implicant, 454–455
charts, 457–459
essential, 458
obtaining directly, 460–467
generating, 455–457
Principle of optimality, 333
Principles, coding, 53
Probabilities, conditional, 341
under conditions, of risk, 327–328
of uncertainty, 329
diagnostic, 343
alternative, 341–344
in dynamic programming, 330–333
in medical diagnosis, 341–348
transition, 330–333
in treatment, 345–348
Problem-solving program, general, 355–362
Procedure, cookbook language, 201, 202
Procedure body, ALGOL, **230**–232
Procedure declaration, ALGOL, **230**–234
for function, 234–235
relation to procedure statement, 231–232
COBOL (**DEFINE** statement), **256**–257
cookbook language, 201, 202

Procedure division, COBOL, 244, 246
Procedure heading, ALGOL, **230**–232
Procedure identifier, ALGOL, 207–**208, 229,** 230
Procedure-name, COBOL, **243,** 246
Procedure statement, ALGOL, 219, **229**–230, 232
as function, 234–235
relation to procedure declaration, 231–232
COBOL (**INCLUDE** statement), 254–**255,** 256
cookbook language, 201
Product, elementary, 451
transposition of Boolean matrices, 476
Product-term, ALGOL, **210**
Profile, symptom-disease, 338, 342
Profit in dynamic programming, 330–333
Program, automatic, 159
automonitor, 161–163
COBOL, 244, 276
debugging, 53, 160–164
definition, **7, 53**
general problem-solving, 355–362
intermediate, 159
(*See also* Programming)
PROGRAM ID clause, COBOL, 277
Program identification, COBOL, **277**
Programmer, definition, **7, 41**
role, 6–7
Programming, to achieve intelligence, 295–382
automatic (*see* Automatic programming)
business logistic computer, 147–148
digital differential analyzer, 130–132
dynamic, 330–333
in medical diagnosis, 344–348
heuristic, 348–355
comparison with translator, 355
references, 379–381
technique, 351–354
linear, 325–326
philosophy of, 54
real-time control computer, 133–135
stages, 53–54, 71–72
for Tabledex, 505–506
Proof, geometric, 350
Propagation of error in polynomial differences, 397
Propositional calculus, 437
in medical diagnosis, 335–339
(*See also* Symbolic logic)
Propositional operations, 437–438
Pseudoinstructions, 298
Pulses, 35–36
Punched cards, 30, 41, 43, 105
(*See also* Marginal-punch cards; Peek-a-boo cards)
Punched paper tape, 30, 40–41, 105
Pure strategy, 329
Push-down list, 178–179, 296–297
translator, 311*n.*, 313
Pushbuttons, 120–122
Progress, heuristic programming criteria, 359
Quine-McCluskey simplification, 454–459
Quotation marks, COBOL, 242–243, 246, 265

R matrix, 479–480
Radioactive tracer study, 11, 12
Radix, 54
ordering by, 512–515
Random numbers, in game theory, 329
in musical creativity, 372–375
Rate equations, 11
Reactions, enzyme, analyzing, 484–486
Read-in, initiating, 122
Read-in routine, initial, 115–116
READ statement, COBOL, 283–285
Read and write instructions, types, 105–106, 115–116
Reading from memory, 47–48
Real-time control computer, 132–137
external-control instruction, 136
flow chart, 135
programming, 133–135
real type, ALGOL, 226, 230
Reasoning in medical diagnosis, 334–335
Receiving-area list, COBOL, **279**
Receiving field, COBOL, 279, 281
Receiving-field list, COBOL, **279**
Recognition, abstraction (pattern), 363–367, 369–371
references, 381–382
Record, COBOL, 244, **246**
format, 260–263
label, 272–275
processing, 259–260, 282–285
storing, 257–260, 262, 268, 270, 272
(*See also* Data-record)
Record area, COBOL, 272
RECORD CONTAINS clause, COBOL, 272
Record description, COBOL, 246, 260–**261,** 262–263
Record-handling statements, COBOL, 283–285
Record-name, COBOL, **272,** 285
Record-name list, COBOL, **272**
Recursion codes, 72–76
in ALGOL, 221
(*See also* Loops)
Recursive compiler, 190–191
Recursive definition, 198
REDEFINES clause, COBOL, 267–268, 271
Redundant codifying, 523–526
definition, **522**
Redundant symptoms, 369
Reel, tape, labeling in COBOL, 272, 274
References, ALGOL, 238–239
automatic programming, 192
automatic-programming translation, 378
Boolean algebra, foundations and axiomatic approach, 493–494
COBOL, 291–294
computer applications, 49–52
computer manuals, 125
computer-systems control, 157
decision theory, 378–379
digital differential analyzer, 156–157

References, error detection and correcting codes, 535–536
heuristic programming, 379–381
information retrieval, 534–535
instruction formats and operation types, 124–125
list processing, 377–378
logical translation of English, 494
medical diagnosis, 379
multiple-component computing systems, 157
numerical analysis, 433–434
pattern recognition and concept formation, 381–382
simplification of Boolean functions, 494
sorting and ordering, 534
symbolic logic, formal theory, 493
Registers, Honeywell 800 computer, 537–540, 544
index, 110
use, in list processing, 298
in push-down lists, 178–179
IBM 7090 computer, 542–544
instruction, 39
lights for contents, 120
manual cleaning, 121
UNIVAC 1103A computer, 541
Reinforcement rules, neuron, 370
Relation, ALGOL, **209**
COBOL, **247**, 253
syntactical, in abstraction, 363–364
Relational operand, COBOL, **247**–248
Relational operation, COBOL, **247**
(*See also* Relational operator)
Relational operator, ALGOL, **206**, 209
COBOL, **242**
(*See also* Relational operation)
Relative addressing (*see* Addressing)
Relative error, 424–426
Relaxation of difference equations, 419–421
Relaxed conditions of searching, 506–510
Remarks, coding sheet, 65
REMARKS clause, COBOL, 277
RENAMES clause, COBOL, 268–270
Repeat instruction, UNIVAC 1103A computer, 541
Replacement, COBOL, **255**
Replacement list, COBOL, **255**
REPLACING, COBOL, 255
Representations of Boolean expressions, hypercube, 446
lattice diagram, 446–447
set, 442–444
Venn diagram, 444, 445
Reservation system, airline, 23–25
Residual, difference equation, 420
digital differential analyzer, 128, 129
Result, 26
Retrieval, information, 25, 26
(*See also* Data processing; Searching)
Retrieval computer, 137–140, 149–150
Return jump instruction, 104, 165
Rewind options, COBOL, **283**
Rhythm, 371, 373–375
Ring addition, 102
Risk, decision under, 325–328
Round numbers, 423–424
Round-off errors, 422
Rounding, 423
Routine, 53
(*See also* Subroutine)
Rule, Simpson's, 406–407, 409–410, 413
trapezoid, 406
Rules, precedence, ALGOL, arithmetic, 212
relational and logical, 209
COBOL, 248
logical, 248
reinforcement, neuron, 370
selection, musical, 372–373
significant figure loss, 426–428
Runge's method for differential equations, 413–415

S (sign), COBOL, 263
Scaling, 69, 99, 131
Scope of local variable, ALGOL, 228
SEAC, 6
Search, conventional, 173–176
(*See also* Searching)
Search pseudoinstruction, 174
Searching, definition, **495**
drum, 149, 150
ingredients, 495
methods, 500–510
with computers, 500–503
manual, 504–506
Tabledex, 502–503
with permanent conditions, 508
with preferential conditions, 508–509
problems in terms of Boolean algebra, 499–500
programming for, 173–176
for r out of n characteristics, 507–508
with relaxed conditions, 506–510
(*See also* Codifying; Data processing; Sorting)
Searching computer, 137–140, 149–150
SELECT, COBOL, 275
Selection, of arguments, 28–30, 33–34, 39
(*See also* Addresses; Addressing; Memory)
rules, musical, 372–373
Selector, in-out, 30, 153
(*See also* Interlocks)
Self-correcting redundant codification, 523–526
Self-organizing nets, 369
Semantic discussions of language, 194, 199–200
Semantic heuristic, 352, 354
Semantic routine, heuristic programming, 354
Semantics, ALGOL, assignment statement (*see* Statement)
conditional statement, 219–221
for statement, 221–225
go to statement, 218–219
procedure declaration, 231–232
procedure statement, 229

Semantics, ALGOL, simple arithmetic expression, modified, for efficient translation, 320–321
for translator, 312
English-Korean, 301–304
of language, 199–200
Semicolon, ALGOL, 205, 216, 230
COBOL, 242, 246, 251, 261, 284
Sending area, COBOL, **279**
Sending field, COBOL, **279**, 281
Sense-indicator instructions, IBM 7090 computer, 544
Sentence, COBOL, 246
English and Korean, **301**
Sequences of decisions, 330–333
in medical diagnosis, 334–335
Serial computers, 38
Series approximation, 393–394
(*See also* Polynomial approximation)
Sets, 442–444
calculus (*see* Boolean algebra)
Sexadecimal, 55, 56
conversion, 55–60
Sharing high-speed memory, 179
Sheffer stroke, 442*n*.
Shift instructions, IBM 7090 computer, 543
types, 103
Sign, COBOL, 263–264
medical, 335*n*.
(*See also* Symptom)
(*See also* Break-point)
Sign bit, 68–69
SIGNED, COBOL, 264
Significant figures, 423–424, 425–429
Simple arithmetic expression, ALGOL, 211–**212**, 213
automatic translation, 311–324
Simple Boolean expression, ALGOL, 209–**210**, 211
Simple designational expression, ALGOL, **215**
Simple tally condition, COBOL, **253**–254
Simple variable, ALGOL, **207**, 208, 230
Simplest product of sums, 451, 489–490
Simplest sum of products, 454
constrained basis, 470–471
methods, decimal, 459–460
Harris-Elbowin, 460–467
Quine-McCluskey, 454–459
Simplification of Boolean functions, references, 494
Simpson's rule for integration, 406–407, 409–410, 413
Simulation, 8
industrial dynamics, 15–19
man-machine, 18–21
neural net, 369
Simulational interpretive routine, 164, 168–172
Simultaneous Boolean equations, 483
Simultaneous linear equations, 384–387
SIZE clause, COBOL, 263, 265, 270
Sort, simple, 507
Sorting, definition, **498**
marginal-punched cards, 504, 532, 533
Sorting, references, 534
Source code list, 306
Source language, 299
Space, memory, sharing, 179
Spare parts inventory, 140–141
Special-purpose computers, 126–157
business and logistics, 140–150
digital differential analyzer, 127–132
information retrieval, 137–140, 149–150
real-time logical control, 132–137
references, 156–157
Specification part, ALGOL, **230**–231
Specifier, ALGOL, **230**
Speed, of computer, 28, 36, 43, 47, 179–181
of information conversion from one medium to another, table, 43
Split tree, 113
Standard address for return jump, 165
Standard basis, 449
STANDARD label-record description option, COBOL, 273
Standard position, elementary data-item, COBOL, 280–281
State, of health, 345–348
of a system, 325
alternative, under risk, 326–327
in dynamic programming, 330–333
Statement, **ADD**, COBOL, **249**–250
ALGOL, 215–225, **219**, 221, 230
labeling, 216
in procedure declaration, 230–231
types, 215
ALTER, COBOL, 250–**251**
assignment, ALGOL, 203–205, **217**–219, 228
automatic translation, 311–324
modified syntax, for efficient translation, 320–321
for translator, 312
COBOL, class, 249–250
automatic-programming language, 197
CLOSE, COBOL, **283**, 284
COBOL, 243, 246
compound, ALGOL, 215–217, 219
COMPUTE, COBOL, **249**
conditional, ALGOL, **219**–221
ALGOL-COBOL differences, 251–252
COBOL, **251**–253
data-handling, COBOL, 278–279
DEFINE, COBOL, **256**–257
DIVIDE, COBOL, **250**
else, COBOL, **251**–252
file-handling, COBOL, 282–285
for, ALGOL, 219, **221**–225
COBOL (*see* **PERFORM**, *below*)
go to, ALGOL, 214, **218**–219
with conditional statement, 220–221
in ***for*** statement, 225
in ***switch*** declaration, 227–228
GO TO, COBOL, 250–**251**, 253
if, ALGOL, **219**
compound, 219
COBOL, **251**–252
INCLUDE, COBOL, 254–**255**, 256
MOVE, COBOL, 278–**279**, 280–282, 285

Statement, **MOVE CORRESPONDING**, COBOL, **279**
MULTIPLY, COBOL, **250**
nonconditional, COBOL, 284
OPEN, COBOL, **283**, 284
operational, COBOL, 247–257, 277
PERFORM, COBOL, **253**–254, 255
procedure, ALGOL, 219, **229**–230, 232
as functions, 234–235
COBOL (*see* **INCLUDE**, *above*)
relation to procedure declaration, 231–232
READ, COBOL, 283–285
record-handling, COBOL, 283–285
SUBTRACT, COBOL, **250**
unconditional, ALGOL, **219**, 220
WRITE, COBOL, 283–285
Statistics, medical diagnosis, 341–342
Step-until, ALGOL, 221, 223–224
Stop instruction, 66, 205*n*.
Storage location, definition, **31**
Stored program (*see* Instruction modification)
Strategy, game theory, 329
Stroke, Sheffer, 442*n*.
Structures, in ALGOL, 204
in a language, 199
Style, musical, 375
Subblocks, ALGOL, 228
Subgoals, heuristic programming, 351–352
Subject, English and Korean, **301**
Sublists, 177, 178
Subroutine, in ALGOL, 228–229, 231–232
name, 229, 231
binomial-coefficient look-up, 78
in COBOL, 254–257
column-adding, 79
data required to use, 81–85
data supplied to, 84, 165
definition, 80
flow chart for setting up, 82
largest number, 67–68
in ALGOL, 221, 223
flow chart, 70
in one-address, 95
in symbolic code, 71
in three-address, 90
in two-address, 92
library, 84–85
missile altitude, 64–66
flow chart, 69
in one-address, 94
in three-address, 89
in two-address, 91
missile trajectory, 72–74
sin x, 79–80
square root, 76
subroutine-modifying, 166–167
syntax-based translator, 299–301, 306–311
table look-up, 77–78, 117
translator, 306–311
in abstraction, 363–365
verb, COBOL, **256**
(*See also* Flow chart; Functions)
Subscript name, COBOL, 253
Subscript tally condition, COBOL, **253**–254
Subscripted variable, ALGOL, 207
array declaration, 226
COBOL, 243, 254, 265–270
(*See also* Data-name)
Substantive-based ordering, 503
SUBTRACT statement, COBOL, **250**
Subtraction, binary and octal, 61–62
Subtraction instructions, types, 101
Successive approximations, 387–393
Sum, elementary, 452
Sum-term, ALGOL, **210**
Superimposed irredundant codification, 526–532, 534
Switch, ALGOL, 230
Switch declaration, ALGOL, **227**–228, 230
Switch designator, ALGOL, **227**–228
Switch list, ALGOL, **227**
Switch sensing, one-shot and on-off, 114
Switches, auxiliary, 104–105, 121
Syllabary characters, 195, 196
Symbolic addresses, translating, 182–185
Symbolic code, 71
compiling, 182–186
Honeywell 800 computer, 540–541
IBM 7090 computer, 543–544
instructions for translator, 311
translating, 189
UNIVAC 1103A computer, 541–542
Symbolic coding, 70–71
Symbolic logic, 436
combined elements, 438
equivalence, 439
implication, 439
tautology, 439–440
elementary elements, 438
formal theory, references, 493
in medical diagnosis, 335–339
propositional operations, 437–438
propositions, 436–437
truth tables, 447–448
Symbols, basic, Boolean algebra, 448*n*.
COBOL, arithmetic, 242, 248
editing, **242**
flow-chart, 69, 74
key, translator, 307, 310
logical operation, 102–103, 438
ALGOL, 206
COBOL, 241–242, 247–248
matrix operation, 474–475
primitive, 350
syntactical-definition, 199
translator, 306
(*See also* Notation)
Symmetry in heuristic programming, 353
Symptom, medical, 335
ease of obtaining, 339
independent, 344
irrelevant, 369
redundant, 369
typical, 368–369
Symptom complex, 337
probability, 343
Symptom-disease complex, 337

Symptom-disease profile, 338
Symptom profile, 338, 342
Synchronization, 36
Syntactic heuristic, 352, 353
Syntactic symmetry, heuristic programming, 353
Syntactical analysis, 295
in abstraction, 363–365
ALGOL, simple arithmetic expression, 213
simple Boolean expression, 210
(*See also* Syntax)
Syntactical definitions of language, 194, 197–199
symbols, 199
Syntactical relations, in abstraction, 363–364
Syntax, automatic-programming language, 197–198
English-Korean, 301–304
house concept, 364
procedure declaration, ALGOL, 230
simple arithmetic expression, ALGOL, modified, for efficient translation, 320–321
for translator, 312
statements (*see* Statement)
translator based on, 298–324
USAGE clause, COBOL, 264
(*See also* Syntactical analysis)
Syntax routine, heuristic programming, 354–355

T, COBOL, 241
(*See also* Logical value)
Table, idealized concept, 367
Table look-up, 77–78, 116–118
Table reference, COBOL, 244
differentiation by, 408
Tabledex, 502–503
manual, computer generation, 505–506
Tables, in COBOL, 243–244, 265–267
of constants, 271, 274
truth, 447–448
Tally, 74
Tally condition, ALGOL, **221**–223
COBOL, simple, **253**–254
subscript, **253**–254
Tape (*see* Magnetic tape; Paper tape)
Target fragments, 306
Target language, 299
instructions for translator, 311
Tautology, 439–440
Taylor series, 402, 412, 413
Temporaries, 66
Term, ALGOL, arithmetic, **212**, 214
Boolean, **209**–211
Terminology, computer, 31–32
Test, medical, 335*n*.
(*See also* Symptom)
THEN, COBOL, 251
Theorems, geometric, 350–355
trigonometric, 355–361
Theories in logical problems, 483
Theta product of Boolean matrices, 493
Threaded lists, 177
push-down lists in, 179
Three-address instruction format, 88
Three-address instruction system, 88–90
example, 537, 540
Threshold, neuron, 369–370
THRU, COBOL, 276
TIMES, COBOL, 242, 248, 253, 265
TO, COBOL, 249–250, 279
TO PROCEED TO, COBOL, 251
Toymaker's problem, 330–333
Transducers, 15
Transfer instruction, 91–93, 167, 170
Honeywell 800 computer, 540
Transformation, Boolean algebraic, 479–482
Transformation operator, heuristic programming, 352
Transformation types, heuristic programming, 355–357
Transition probability, 330–333
in treatment, 345–348
Transition value, dynamic programming, 330–333
Translating computing unit, 151
Translation, automatic-programming-language, 298–324
English-Korean, 301–305
human language, 302
logical, of English, 444–446
references, 494
Translator, syntax-based, 298–324
in abstraction, 363–365
comparison with heuristic program, 355
Translator compiler, 182–185
Transpose of Boolean matrix, 474
Transposition, ordering by, 511–513
Trapezoidal rule for integration, 406
ALGOL code, 233
Treatment decisions, alternative, 325–330
sequences, 330–333
Tree, split, 113
Trigonometric theorems, 355–361
True (*see* Logical value)
Truncated errors, 421–422
Truth tables, 447–448
Two-address instruction format, 90
Two-address instruction system, 90–93
example, 541–542
simulating one-address in, 168–172
Type, ALGOL, **226**, 227, **230**–231, 234
COBOL item, 261
(*See also* **CLASS**)
heuristic, 355–357
label-record, COBOL, 271–**273**, 274
searching on, 174–175
threaded-list, 177
transformation, heuristic programming, 355–357
usage, COBOL, **264**
Type or branch link, list, 297
Type declaration, ALGOL, **226**–228
Type-face chain, 45
Type-face drum, 44

Uncertainty, decision under, 325, 328–330
Unconditional statement, ALGOL, **219,** 220
Undetermined coefficients, function evaluation by, 409–410
Union of sets, 443
Unitary matrix equations, 476–478
UNIVAC 1103A scientific computer, 541–542
UNIVAC X-3 code, 96
Unnormalized, 99
Unsigned integer, ALGOL or COBOL, 198–**199,** 215
Unsigned number, ALGOL, 212
(*See also* Decimal number)
until, ALGOL, 221, 223–224
UNTIL, COBOL, 253
USAGE, clause, COBOL, **264,** 270
Usage type, COBOL, **264**

Value, absolute, in ALGOL, 234
alternate-treatment-decision, under conditions, of certainty, 325–326
of risk, 327–328
of uncertainty, 328, 329
condition-name, COBOL, 264–265
(*See also* **VALUE** clause)
condition-variable, COBOL, 265
data-name, COBOL, 242, 256
initial, 270
decision-state pair, 325
expression, automatic-programming language, 197
logical, ALGOL, **206,** 209
in Boolean expression, 208–211
COBOL, **241,** 247
numerical, in ALGOL arithmetic expression, 211, 213–214
subscripted variable, COBOL, 266, 267
transition, 330–333
threaded-list type, 177
VALUE clause, COBOL, 264–265, 270–271
Value-literal, COBOL, 265
VALUE OF clause, COBOL, 273–274
Value part, ALGOL, **230**–232
Variable, ALGOL, 206–**207,** 208, 209, 212, 215, 217, 221, 226, 227, 229
automatic-programming language, 197
Boolean algebra, 448*n*.
transformation, 479–482
COBOL (*see* Data-name)
controlled, ALGOL, 225
dependent and independent, ALGOL, **229**–231, 235
COBOL, 255–256
independent, ALGOL, **208**
local and nonlocal, ALGOL, 228
Variable, logical, ALGOL, 209
simple, ALGOL, **207,** 208, 230
subscripted, ALGOL, 207
in ***array*** declaration, 226
COBOL, 254, 265–270
Variable connector, 81
in ALGOL, 227–228
Variable list, ALGOL, **226, 230**
Variable-word addressing, 106–108
VARYING, COBOL, 253
Vector in ALGOL, 226
Venn diagram, 444, 445
Verb, COBOL, **243,** 249
(*See also* Statement)
English and Korean, **301**
Verb-name, COBOL, **256**
Verb structure, COBOL, **256**
Verb subroutine, COBOL, **256**

Well-formed formulas, in geometry, 350
in heuristic programming, 357
while, ALGOL, 221, 224–225
Whirlwind, 6
WITH, COBOL, 283
Word, automatic-programming language, 195–197
COBOL, **242**–244
kinds, 242
as literals, 247
computer, 31
end-of-data, 176, 177
form, 35–36
inserting single, 121
interpretation, 32, 35
labeling, 173
length, 35
special read-only, 113
Word block, 105
Word format, 39, 62–66, 88, 90, 95–96
one-address, 94
IBM 7090 computer, 542
simulated, 169
three-address, Honeywell 800 computer, 538
two-address, 92
UNIVAC 1103A computer, 541
Working-storage area, COBOL, 260, 268, 270
transfer to or from, 278–279
WRITE statement, COBOL, 283–285
Writing a code, steps in, 53–54, 71–72

X (alphanumerical character), COBOL, 263
X-ray diffraction, 10

ZERO SUPPRESS clause, COBOL, 282